CSCE 585: Machine Learning Systems

Lecture 5: LLM Inference (LLMOps)

Pooyan Jamshidi

Fall 2024 | Course Website: https://pooyanjamshidi.github.io/mls/

UNIVERSITY OF South Carolina

LLM Inference Basics

What is LLM inference?

- LLM inference refers to using trained LLMs, such as GPT-4, Llama 4, and DeepSeek-V3, to generate meaningful outputs from user inputs, typically provided as natural language prompts.
- During inference, the model processes the prompt through its vast set of parameters to generate responses like text, code snippets, summaries, and translations.

Real-world examples

- **Customer support chatbots:** Generating personalized, contextually relevant replies to customer queries in real-time.
- **Writing assistants:** Completing sentences, correcting grammar, or summarizing long documents.
- **Developer tools:** Converting natural language descriptions into executable code.
- **AI agents:** Performing complex, multi-step reasoning and decision-making processes autonomously.

Why should I care about LLM inference?

- You might think: I’m just using OpenAI’s API. **Do I really need to understand inference?**
- Serverless APIs like OpenAI, Anthropic, and others make inference look simple. You send a prompt, get a response, and **pay by the token**.
- The **infrastructure**, **model optimization**, and **scaling** are all hidden from view.
- As your application grows, you'll eventually run into limits (e.g., cost, latency, customization, or compliance) that serverless APIs can’t fully address. That’s when teams start exploring **hybrid** or **self-hosted solutions**.

Why should I care about LLM inference?

- If you're a **developer**: Inference is becoming as fundamental as databases or APIs in modern AI application development. Knowing how it works helps you design faster, cheaper, and more reliable systems. Poor inference implementation can lead to slow response time, high compute costs, and a poor user experience.
- If you're a **technical leader**: Inference efficiency directly affects your bottom line. A poorly optimized setup can cost 10× more in GPU hours while delivering worse performance. Understanding inference helps you evaluate vendors, make build-vs-buy decisions, and set realistic performance goals for your team.
- If you're just **curious about AI**: Inference is where the magic happens. Knowing how it works helps you separate AI hype from reality and makes you a more informed consumer and contributor to AI discussions.

What is the difference between LLM training and inference?

Training: Building the model's understanding

- It is about **teaching the model how to recognize patterns** and make accurate predictions. This is done by exposing the model to vast amounts of data and adjusting its parameters based on the data it encounters.
- Common techniques used in LLM training include:
  - **Supervised learning**: Show the model examples of inputs paired with the correct outputs.
  - **Reinforcement learning**: Allow the model to learn by trial and error, optimizing based on feedback or rewards.
  - **Self-supervised learning**: Learn by predicting missing or corrupted parts of the data, without explicit labels.

What is the difference between LLM training and inference?

Inference: Using the model in real-time

- LLM inference means applying the trained model to new data to make predictions.
- Inference compute needs are ongoing and can become very high, especially as user interactions and traffic grow.
- While each inference step may be smaller than training in isolation, the cumulative demand over time can lead to significant operational expenses.

How does LLM inference work?

What are tokens and tokenization?

- During inference, an LLM generates text one token at a time, using its internal attention mechanisms and knowledge of previous context.
- A token is the smallest unit of language that LLMs use to process text.
- It can be a word, subword, or even a character, depending on the tokenizer.
- Each LLM has its own tokenizer, with different tokenization algorithms.
- Tokenization is the process of converting input text (like a sentence or paragraph) into tokens.
- The tokenized input is then converted into IDs, which are passed into the model during inference.

`BentoML supports custom LLM inference.`

```
Tokens: "B", "ento", "ML", " supports", " custom", " L", "LM", " inference", "."

Token IDs: [33, 13969, 4123, 17203, 2602, 451, 19641, 91643, 13]
```

The two phases of LLM inference

prefill

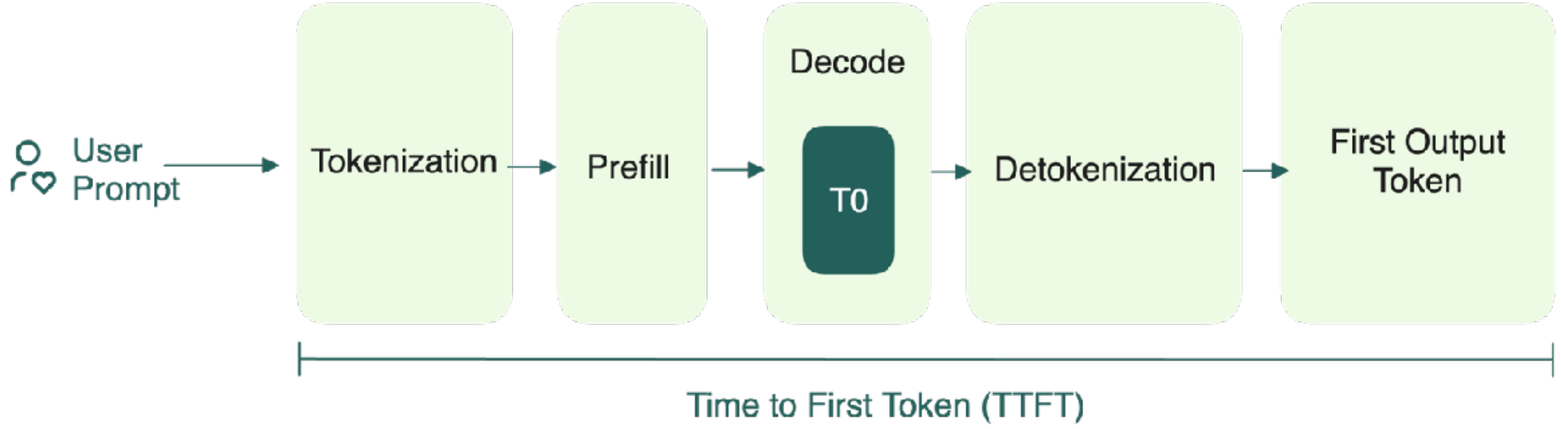
- When a user sends a query, the LLM's tokenizer converts the prompt into a sequence of tokens. The prefill phase begins after tokenization:

    1. These tokens (or token IDs) are embedded as numerical vectors that the LLM can understand.

    2. The vectors pass through multiple transformer layers, each containing a self-attention mechanism. Here, query (Q), key (K), and value (V) vectors are computed for each token. These vectors determine how tokens attend to each other, capturing contextual meaning.

    3. As the model processes the prompt, it builds a KV cache to store the key and value vectors for every token at every layer. It acts as an internal memory for faster lookups during decoding.

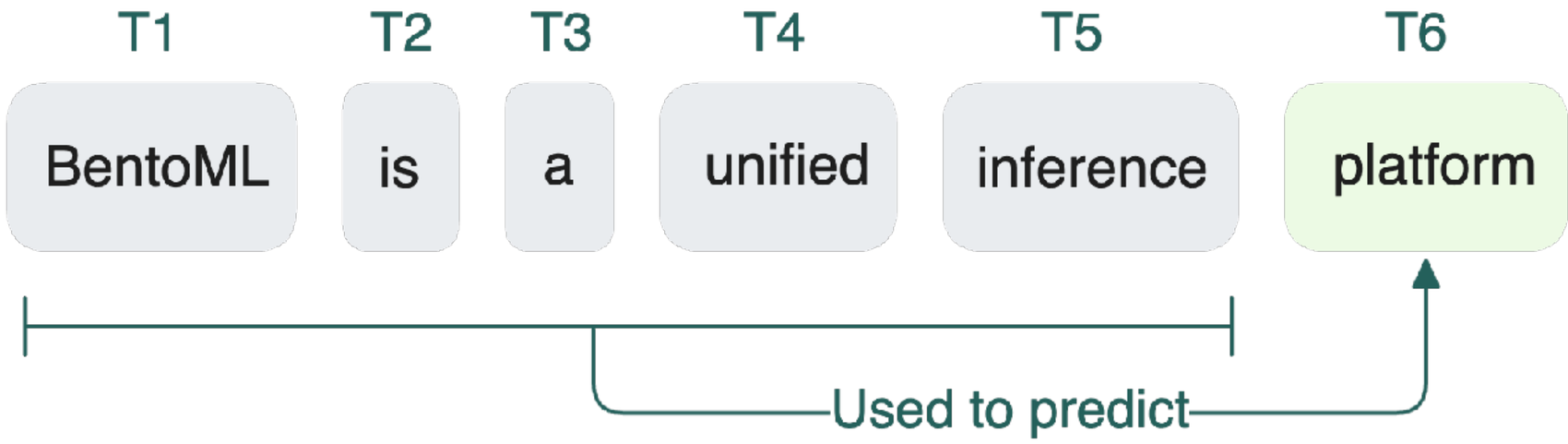
- LLM can process all tokens simultaneously through highly parallelized matrix operations, particularly in the attention computations.



The two phases of LLM inference

Decode

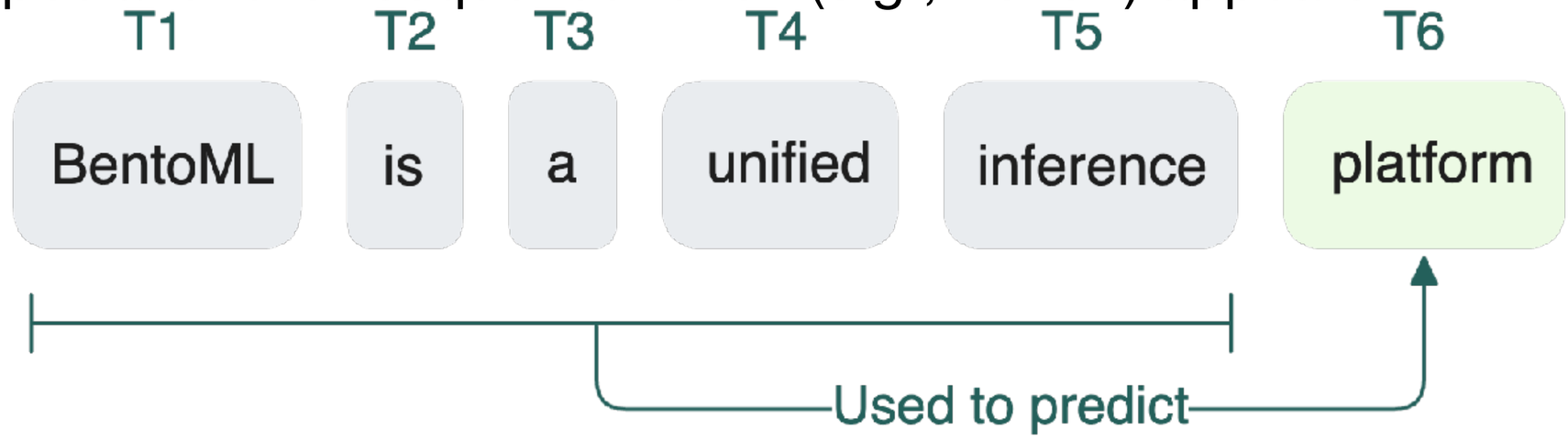
- After prefill, the LLM enters the decode stage where it generates new tokens sequentially, one at a time.
- For each new token, the model samples from a probability distribution generated based on the prompt and all previously generated tokens. This process is autoregressive, meaning tokens T_0 through T_{n-1} are used to generate token T_n, then T_0 through T_n to generate T_{n+1}, and so on.



The two phases of LLM inference

Decode

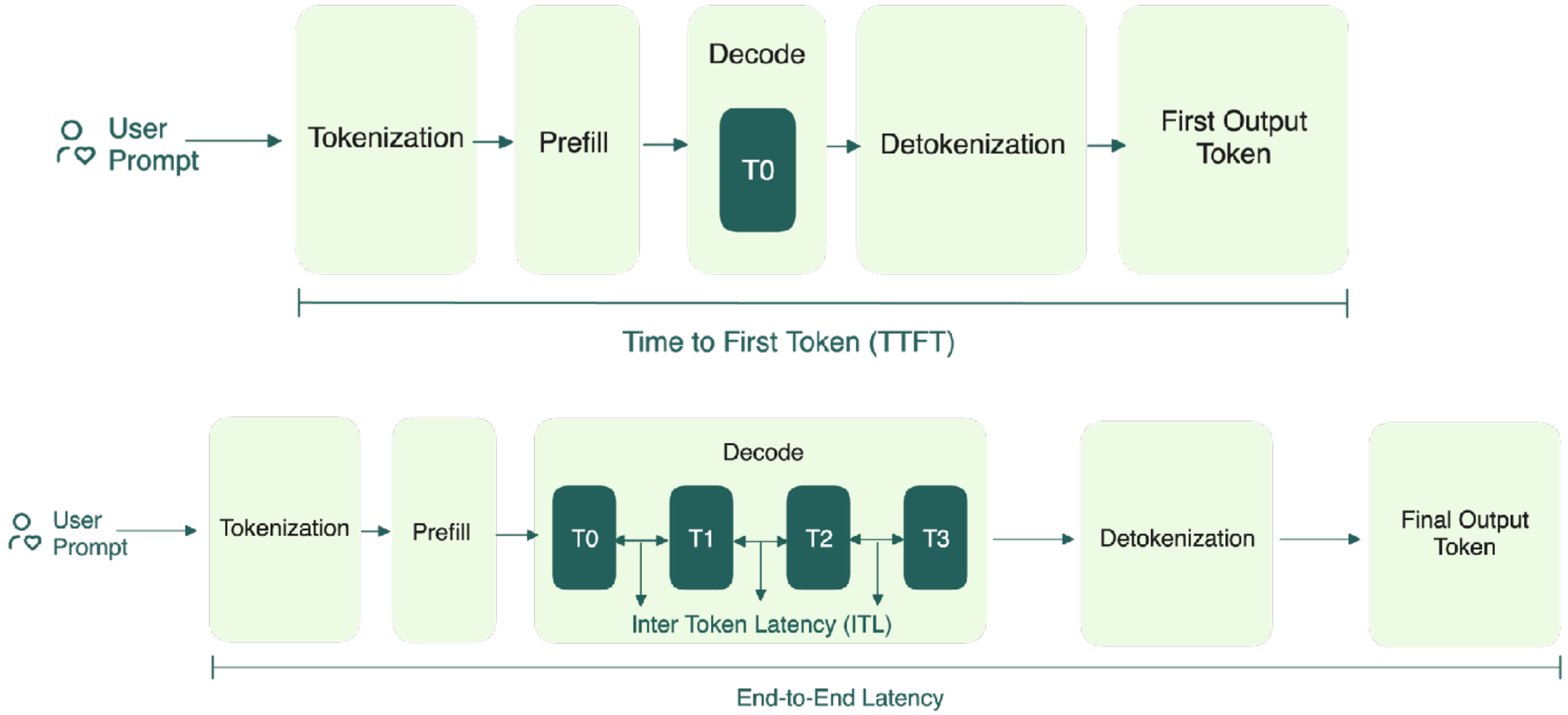
- Each newly generated token is appended to the growing sequence. This autoregressive loop continues until:
  - A maximum token limit is reached,
  - A stop word is generated,
  - Or a special end-of-sequence token (e.g., <end>) appears.



Compute vs Memory in LLM Inference

- The prefill stage is **compute-bound** and often saturates GPU utilization. The actual utilization depends on factors like sequence length, batch size, and hardware specifications.
- Compared with prefill, decode is more **memory-bound** because it frequently reads from the growing KV cache. KV caching stores these key and value matrices in memory so that, during subsequent token generation, the LLM only needs to compute the keys and values for the new tokens rather than recomputing everything from scratch.
- This KV caching mechanism significantly speeds up inference by avoiding redundant computation. However, it comes at the cost of increased memory consumption, since the cache grows with the length of the generated sequence.

Key performance metrics in LLM inference



Note: Detokenization happens after each decode step. Each token (T0, T1, T2) is detokenized and output sequentially, not just the final one (T3).

Collocating prefill and decode

- Traditional LLM serving systems typically run both the prefill and decode phases on the same hardware.
- However, this setup introduces several challenges:
  - One major issue is the interference between the prefill and decode phases, as they cannot run fully in parallel.
  - In production, multiple requests can arrive at once, each with its own prefill and decode stages that overlap across different requests.
  - However, only one phase can run at a time. When the GPU is occupied with compute-heavy prefill tasks, decode tasks must wait, increasing token latency, and vice versa. This makes it difficult to schedule resources for both phases.

Where is LLM inference run?

- When deploying LLMs into production, choosing the right hardware is crucial. Different hardware types offer varied levels of performance and cost-efficiency.
- **CPUs** are widely available and suitable for running small models or serving infrequent requests. For production-grade LLM inference, especially with larger models or high request volumes, CPUs often fall short in both latency and throughput.
- The architecture of **GPUs** is optimized for matrix multiplication and tensor operations, which are core components of transformer-based models.
- **TPUs** are designed from the ground up for tensor operations — the fundamental math behind neural networks.

Choosing the deployment environment

- **Cloud**: The cloud is the most popular environment for LLM inference today. It offers on-demand access to high-performance GPUs and TPUs, along with a rich ecosystem of managed services, autoscaling, and monitoring tools.
- **On-Prem**: On-premises deployments means running LLM inference on your own infrastructure, typically within a private data center. It offers full control over data, performance, and compliance, but requires more operational overhead.
- **Edge**: In edge deployments, the model runs directly on user devices or local edge nodes, closer to where data is generated. This reduces network latency and increases data privacy, especially for time-sensitive or offline use cases. Edge inference usually uses smaller, optimized models due to limited compute resources.

Serverless vs. Self-hosted LLM inference

Serverless LLM inference

- Serverless inference services, provided by companies like OpenAI, Anthropic, and other hosted API providers, simplify application development significantly. They manage everything for you, letting you pay per use with no infrastructure overhead.
  - **Ease of use:** You can get started quickly with minimal setup — just use an API key and a few lines of code. There is no need to manage hardware, software environments, or complex scaling logic.
  - **Rapid prototyping:** It is perfect for testing ideas quickly, building demos, or internal tooling without infrastructure overhead.
  - **Hardware abstraction:** Self-hosting LLMs at scale usually requires high-end GPUs (such as NVIDIA A100 or H100). Serverless APIs abstract these hardware complexities, allowing you to avoid GPU shortages, quota limits, and provisioning delays.

Serverless vs. Self-hosted LLM inference

Item	Serverless APIs	Self-hosted inference
Ease of Use	✅ High (simple API calls)	⚠️ Lower (requires LLM deployment and maintenance)
Data Privacy & Compliance	⚠️ Limited	✅ Full control
Customization	⚠️ Limited	✅ Full flexibility
Cost at Scale	⚠️ Higher (usage-based, may rise significantly)	✅ Potentially lower (predictable, optimized infrastructure)
Hardware Management	✅ Abstracted away	⚠️ Requires GPU setup & maintenance

Getting Started

Choosing the right model

Base models

- Base models, also called foundation models, are the starting point of most LLMs. They are typically trained on a massive corpus of text data through unsupervised learning, which does not require labeled data.

- During this initial phase, known as pretraining, the model learns general language patterns, such as grammar, syntax, semantics, and context. It becomes capable of predicting the next word (or token) and can perform simple few-shot learning (handling a task after seeing just a few examples). However, it does **not yet understand how to follow instructions** and **is not optimized for specific tasks out of the box**.

Choosing the right model

Instruction-tuned models

- Instruction-tuned models are built on top of base models. After the initial pretraining phase, these models go through a second training stage using datasets made up of instructions and their corresponding responses.
  - “Summarize this article.”
  - “Explain how LLM inference works.”
  - “List pros and cons of remote work.”

Choosing the right model

Mixture of Experts models

- Mixture of Experts (MoE) models, such as DeepSeek-V3, take a different approach from traditional dense models. Instead of using all model parameters for every input, they contain multiple specialized sub-networks called experts, each focus on different types of data or tasks.

- During inference, only a subset of these experts is activated based on the characteristics of the input. This selection mechanism enables the model to route computation more selectively—engaging different experts depending on the content or context. As a result, MoE models achieve greater scalability and efficiency by distributing workload across a large network while keeping per-inference compute costs manageable.

Choosing the right model

Combining LLMs with other models

- **Small Language Models (SLMs).** Used for lightweight tasks where latency and resource constraints matter. They can serve as fallback models or on-device assistants that handle basic interactions without relying on a full LLM.
- **Embedding models.** They transform inputs (e.g., text, images) into vector representations, making them useful for semantic search, RAG pipelines, recommendation systems, and clustering.
- **Image generation models.** Models like Stable Diffusion generate images from text prompts. When paired with LLMs, they can support more advanced text-to-image workflows such as creative assistants, content generators, or multimodal agents.
- **Vision language models (VLMs).** Models such as NVLM 1.0 and Qwen2.5-VL combine visual and textual understanding, supporting tasks like image captioning, visual Q&A, or reasoning over screenshots and diagrams.
- **Text-to-speech (TTS) models.** They can convert text into natural-sounding speech. When integrated with LLMs, they can be used in voice-based agents, accessible interfaces, or immersive experiences.

LLM fine-tuning

- Fine-tuning is one of the most effective ways to adapt an LLM for a specific use case. It continues the training process on a pre-trained model using new, **task-specific data**. This can involve updating the **entire model** or just **specific layers**.

- A key driver behind fine-tuning is efficiency. Instead of training a model from scratch (which is extremely resource-intensive), it's far easier and more cost-effective to build on top of a base model that has already learned general language patterns from massive datasets. Fine-tuning sharpens those broad capabilities for your **particular task**.

LLM fine-tuning

Examples

- **Domain expertise:** Adapting a model for legal, medical, or programming-related tasks.
- **Instruction following:** Ensuring the model adheres to specific formats, tones, or styles in its responses.
- **Safety and alignment:** Reinforcing how the model handles sensitive or high-risk prompts.

Choosing the right inference framework

Inference frameworks and tools

- **vLLM**. A high-performance inference engine optimized for serving LLMs. It is known for its efficient use of GPU resources and fast decoding capabilities.
- **SGLang**. A fast serving framework for LLMs and vision language models. It makes your interaction with models faster and more controllable by co-designing the backend runtime and frontend language.
- **LMDeploy**. An inference backend focusing on delivering high decoding speed and efficient handling of concurrent requests. It supports various quantization techniques, making it suitable for deploying large models with reduced memory requirements.
- **TensorRT-LLM**. An inference backend that leverages NVIDIA's TensorRT, a high-performance deep learning inference library. It is optimized for running large models on NVIDIA GPUs, providing fast inference and support for advanced optimizations like quantization.
- **Hugging Face TGI**. A toolkit for deploying and serving LLMs. It is used in production at Hugging Face to power Hugging Chat, the Inference API and Inference Endpoint.

Choosing the right inference framework

Edge inference frameworks

- **llama.cpp**. A lightweight inference runtime for LLMs, implemented in plain C/C++ with no external dependencies. Its primary goal is to make LLM inference fast, portable, and easy to run across a wide range of hardware. Despite the name, llama.cpp supports far more than just Llama models. It supports many popular architectures like Qwen, DeepSeek, and Mistral. The tool is ideal in low-latency inference and performs well on consumer-grade GPUs.

- **MLC-LLM**. An ML compiler and high-performance deployment engine for LLMs. It is built on top of Apache TVM and requires compilation and weight conversion before serving models. MLC-LLM can be used for a wide range of hardware platforms, supporting AMD, NVIDIA, Apple, and Intel GPUs across Linux, Windows, macOS, iOS, Android, and web browsers.

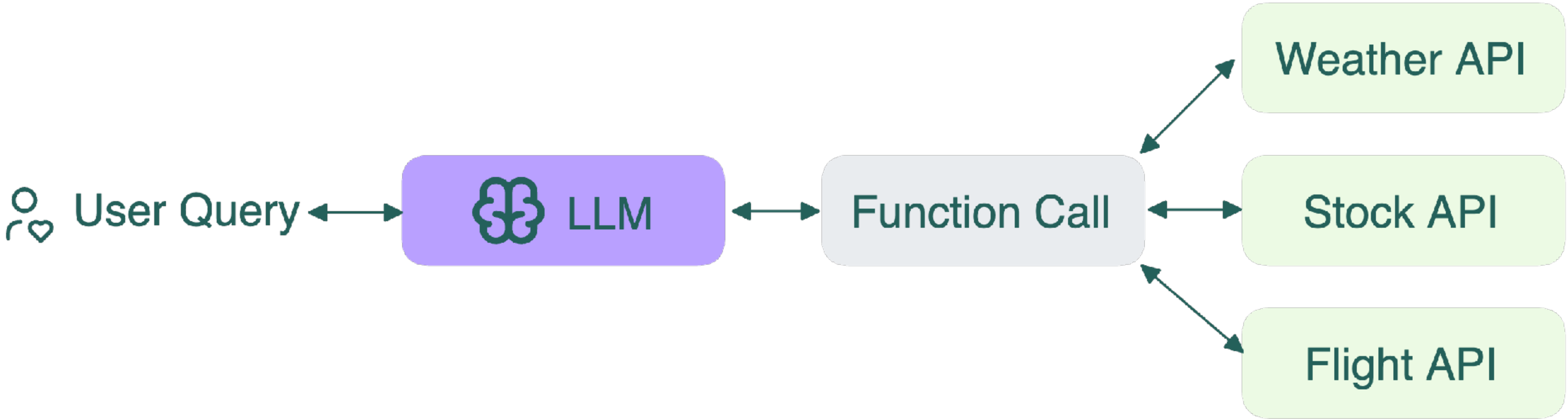
- **Ollama**. A user-friendly local inference tool built on top of llama.cpp. It's designed for simplicity and ease of use, ideal for running models on your laptop with minimal setup. However, Ollama is mainly used for single-request use cases. Unlike runtimes like vLLM or SGLang, it doesn't support concurrent requests. This difference matters since many inference optimizations, such as paged attention, prefix caching, and dynamic batching, are only effective when handling multiple requests in parallel.

Tool integration

Function calling

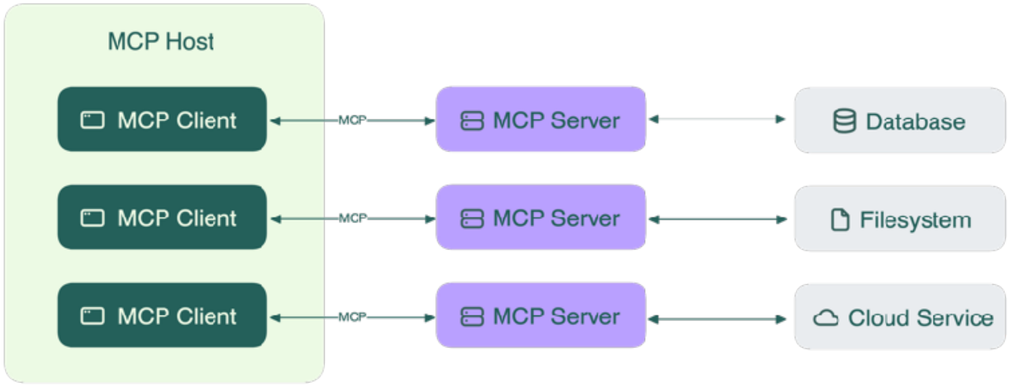
Here is a specific example:

- **You ask**: "What's the current price of Apple stock?"
- **LLM thinks**: "I need current stock data, so I'll use my stock price function"
- **LLM calls**: `get_stock_price("AAPL")`
- **Function returns**: "$195.25"
- **LLM responds**: "The current price of Apple stock is $195.25"



Tool integration

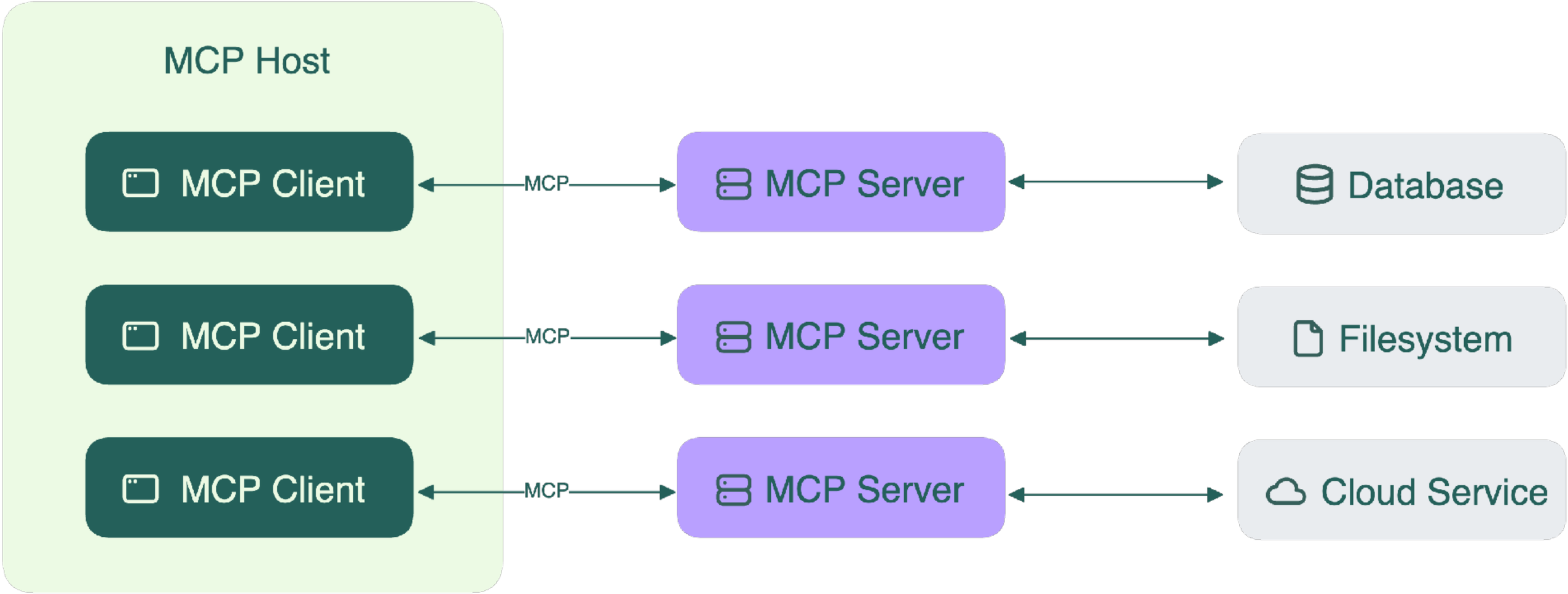
Model Context Protocol



- **MCP host**: This is where the AI assistant lives. It could be a chat application like Claude Desktop, an IDE code assistant, or any other AI-powered application. The host can contain one or multiple MCP clients.
- **MCP clients**: A client is the low-level implementation inside the host that maintains one-to-one links with MCP servers.
- **MCP servers**: The connectors that expose different capabilities and data sources. Each server can connect to various backends like databases, third-party APIs, GitHub repositories, local files, or any other data source. Multiple servers can be running simultaneously on your local machine or connected to remotes services.
- **MCP protocol**: This is the transport layer that enables communication between the host and servers, regardless of how many servers are connected.

Tool integration

Model Context Protocol



Example

1. The MCP host makes a request through the MCP protocol
2. The appropriate MCP server receives the request
3. The server connects to the actual data source (database, API, file system, etc.)
4. The server processes the request and returns the data back through the protocol
5. The AI assistant receives the information and can use it in its response

Infrastructure and operations

What is LLM inference infrastructure?

- LLM inference infrastructure encompasses the **systems** and **workflows** needed to run LLM inference **reliably** and **cost-effectively** in **production**. It includes everything from hardware provisioning to software coordination and operational monitoring.
  - **Hardware provisioning:** Access to high-performance compute resources like GPUs and TPUs.
  - **Orchestration:** Tools that manage resource allocation, scale workloads dynamically, and manage model versions across multiple environments.
  - **Observability systems:** Logging, monitoring, and tracing tools that offer insight into performance metrics such as GPU utilization, latency, throughput, and failure rates.
  - **Operational procedures:** Standardized workflows and automation that enable teams to deploy updates, enforce access control, handle failures, and ensure high availability. As inference demand scales, having repeatable, efficient operations becomes critical to managing growing workloads.

Challenges in building infrastructure for LLM inference

Fast scaling

- That demand is often **bursty**, **hard to predict**, and unforgiving of latency or downtime.
- This means the system needs to scale up quickly during traffic spikes and scale down to zero when idle to save costs.
  - **Over-provisioning:** Wasted GPU capacity, high idle costs.
  - **Under-provisioning:** Dropped requests, latency spikes, and poor user experience.
  - **Inflexible budgets:** Rigid spending that doesn't adapt to real usage patterns.

The cold start problem

- In the context of deploying LLMs in containers, a cold start occurs when a Kubernetes node has never previously run a given deployment.
- As a result, the container image is not cached locally, and all image layers must be pulled and initialized from scratch.

The cold start problem

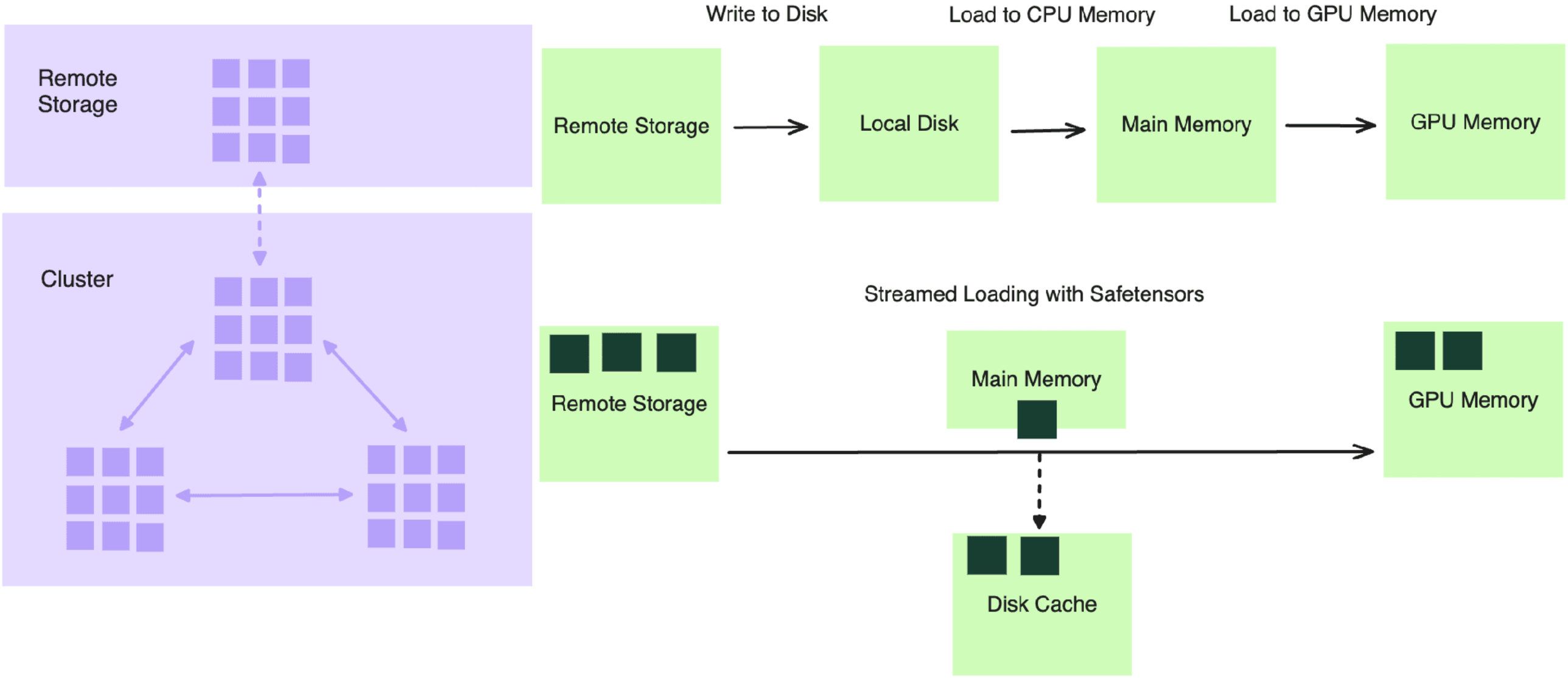
- **Cloud provisioning:** This step involves the time it takes for the cloud provider to allocate a new instance and attach it to the Kubernetes cluster. Depending on the instance type and availability, this can take anywhere from 30 seconds to several minutes, or even hours for high-demand GPUs like Nvidia A100 and H100.

- **Container image pulling:** LLM images are significantly larger and more complex than typical Python job images, due to numerous dependencies and custom libraries. Despite claims of multi-gigabit bandwidth by cloud providers, actual image download speeds are often much slower. As a result, pulling images can take three to five minutes.

- **Model loading.** The time required to load the model depends heavily on its size. LLMs introduce significant delays due to their billions of parameters.

Key bottlenecks in model loading

- **Slow downloads from model hubs:** Platforms like Hugging Face are not optimized for high-throughput, multi-part downloads, making the retrieval of large model files time-consuming.
- **Sequential data flow:** Model files are transferred through multiple hops: remote storage → local disk → memory → GPU. This is minimal or no parallelization between them. Each step adds latency, particularly for large files that are difficult to cache or stream.
- **Lack of on-demand streaming:** Model files must be fully downloaded and written to disk before inference can begin. This introduces additional I/O operations and delays startup.


Remote Storage
Cluster
Write to Disk
Load to CPU Memory
Load to GPU Memory
Remote Storage
Local Disk
Main Memory
GPU Memory
Streamed Loading with Safetensors
Remote Storage
Main Memory
GPU Memory
Disk Cache

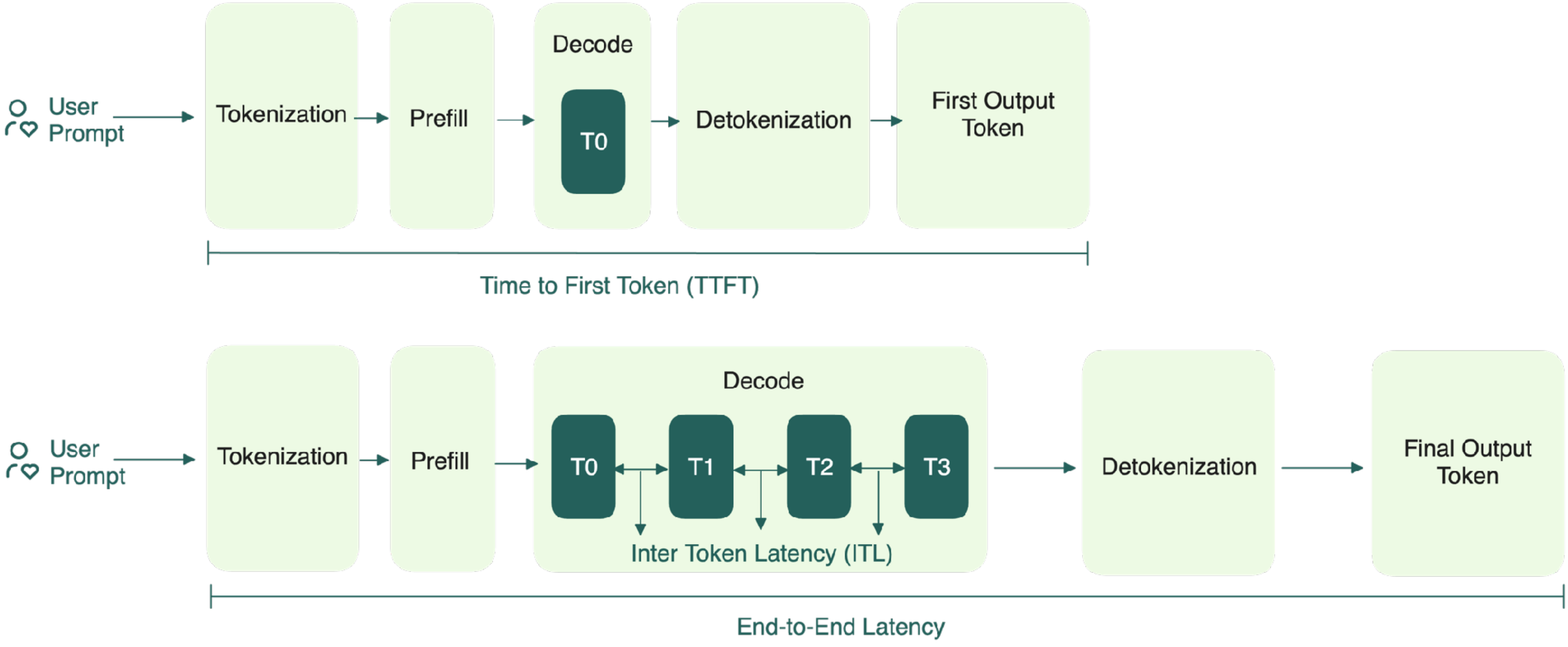
Inference optimization

Key metrics for LLM inference

Latency

- **Time to First Token (TTFT):** The time it takes to generate the first token after sending a request. It reflects how fast the model can start responding.
- **Time per Output Token (TPOT):** Also known as Inter-Token Latency (ITL), TPOT measures the time between generating each subsequent token. A lower TPOT means the model can produce tokens faster, leading to higher tokens per second. In streaming scenarios where users see text appear word-by-word (like ChatGPT's interface), TPOT determines **how smooth the experience feels**. The system should ideally keep up with or exceed human reading speed to ensure a smooth experience.
- **Token Generation Time:** The time between receiving the first and the final token. This measures how long it takes the model to stream out the full response.
- **Total Latency (E2EL):** The time from sending the request to receiving the final token on the user end.

```
Total Latency = TTFT + Token Generation Time
```



Note: Detokenization happens after each decode step. Each token (T0, T1, T2) is detokenized and output sequentially, not just the final one (T3).

Understanding mean, median, and P99 latency

- **Mean (Average):** This is the sum of all values divided by the number of values. Mean gives a general sense of average performance, but it can be skewed by extreme values (outliers). For example, if the TTFT of one request is unusually slow, it inflates the mean.
- **Median:** The middle value when all values are sorted. Median shows what a "typical" user experience. It’s more stable and resistant to outliers than the mean. If your median TTFT is 30 seconds, most users are seeing very slow first responses, which might be unacceptable for real-time use cases.
- **P99 (99th Percentile):** The value below which 99% of requests fall. P99 reveals worst-case performance for the slowest 1% of requests. This is important when users expect consistency, or when your SLAs guarantee fast responses for 99% of cases. If your P99 TTFT is nearly 100 seconds, it suggests a small but significant portion of users face very long waits.

Understanding mean, median, and P99 latency

- **Mean** helps monitor trends over time.
- **Median** reflects the experience of the majority of users.
- **P99** captures tail latency, which can make or break user experience in production.

Key metrics for LLM inference

Throughput

Throughput describes how much work an LLM can do within a given period. High throughput is essential when serving many users simultaneously or processing large volumes of data.

- **Requests per Second (RPS)**: This metric captures how many requests the LLM can successfully complete in one second. It's calculated as:

```
Requests per second = Total completed requests / (T1 - T2)
```

> **NOTE**
>
> Here, T1 and T2 mark the time window in seconds.

RPS gives a general sense of how well the LLM handles concurrent requests. However, this metric alone doesn't capture the complexity or size of each request. For example, generating a short greeting like `"Hi there!"` is far less demanding than writing a long essay.

Factors that impact RPS:

  - Prompt complexity and length
  - Model size and hardware specifications
  - Optimizations (e.g., batching, caching, inference engines)
  - Latency per request

Key metrics for LLM inference

Throughput

Throughput describes how much work an LLM can do within a given period. High throughput is essential when serving many users simultaneously or processing large volumes of data.

- **Tokens per Second (TPS)**: This metric provides a finer-grained view of throughput by measuring how many tokens are processed every second across all active requests. It comes in two forms:
  - **Input TPS**: How many input tokens the model processes per second.
  - **Output TPS**: How many output tokens the model generates per second.

  Understanding both metrics helps you identify performance bottlenecks based on the nature of your inference workload. For example:

  - A summarization request that includes long documents (e.g., 2,000-token inputs) cares more about input TPS.
  - A chatbot that generates long replies from short prompts (e.g., 20-token prompt → 500-token response) depends heavily on output TPS.

  When reviewing benchmarks or evaluating LLM performance, **always check whether TPS metrics refer to input, output, or a combined view**. They highlight different strengths and limitations depending on the use case.

  Factors that impact TPS:

  - Batch size (larger batches can increase TPS until saturation)
  - KV cache efficiency and memory usage
  - Prompt length and generation length
  - GPU memory bandwidth and compute utilization

Latency vs. throughput tradeoffs

Goal	Implication
Maximize throughput (TPS/MW)	Focus on serving as many tokens per watt as possible. This usually means using larger batch sizes and shared compute resources. However, it can slow down responses for individual users.
Minimize latency (TPS per user)	Focus on giving each user a fast response (low TTFT). This often involves small batches and isolated compute resources, but it means you'll use GPUs less efficiently.
Balance of both	Some systems aim for a dynamic balance. They tune resource usage in real time based on workload, user priority, and app requirements. This is ideal for serving diverse applications with different SLOs.

optimization strategies

Inference optimization

For self-hosting open-source or custom models

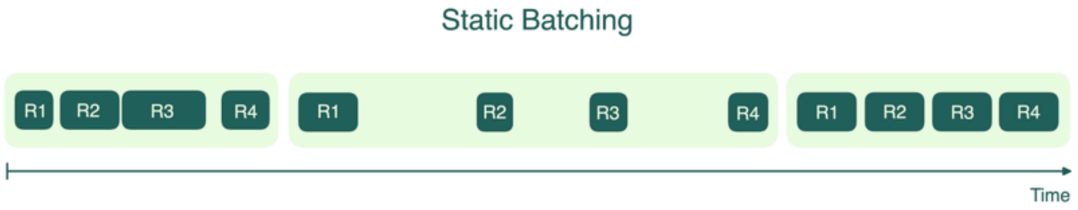
- Static, dynamic, and continuous batching
- PagedAttention
- Speculative decoding
- Prefill-decode disaggregation
- Prefix caching
- Prefix-aware routing
- KV cache utilization-aware load balancing
- KV cache offloading
- Data, tensor, pipeline, expert, and hybrid parallelisms
- Offline batch inference

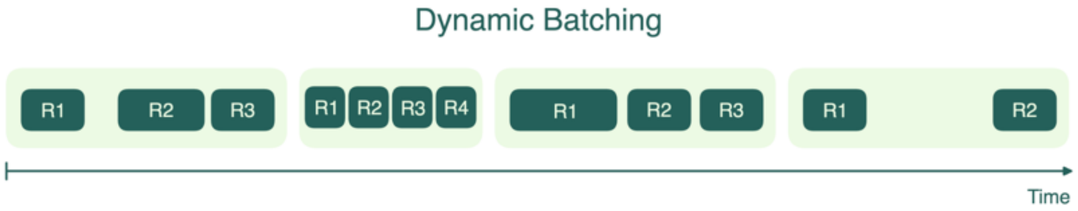
Static, dynamic and continuous batching

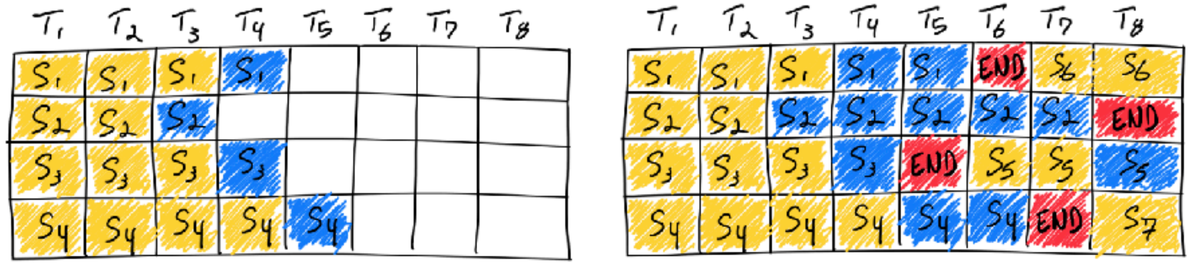
Instead of processing each request individually, batching them together allows you to use the same loaded model parameters across multiple requests, thus dramatically improving throughput.

Static batching

The simplest form of batching is **static batching**. Here, the server waits until a fixed number of requests arrive and then processes them together as a single batch.



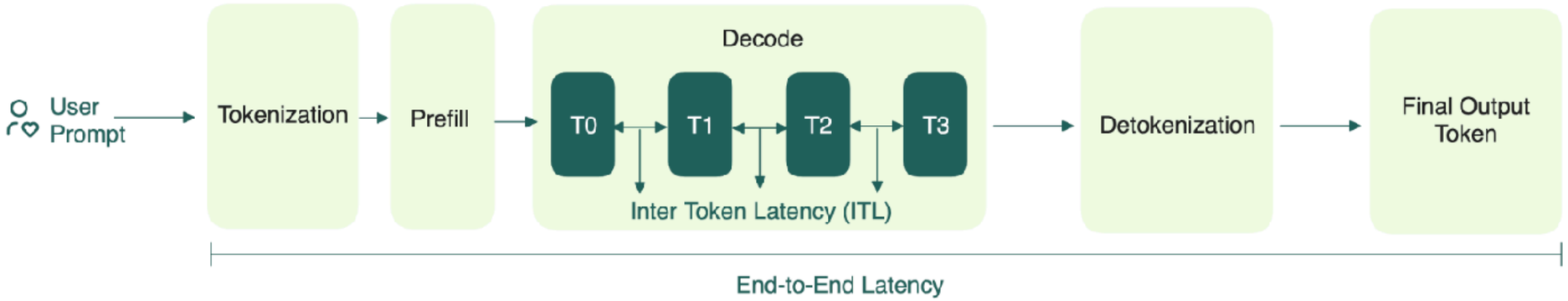




Generating seven sequences with continuous batching. On the first iteration (left), each sequence generates a token (blue) from its prompt (yellow). Over time (right), sequences complete at different iterations by emitting an end-of-sequence token (red), at which point new sequences are inserted. Image source

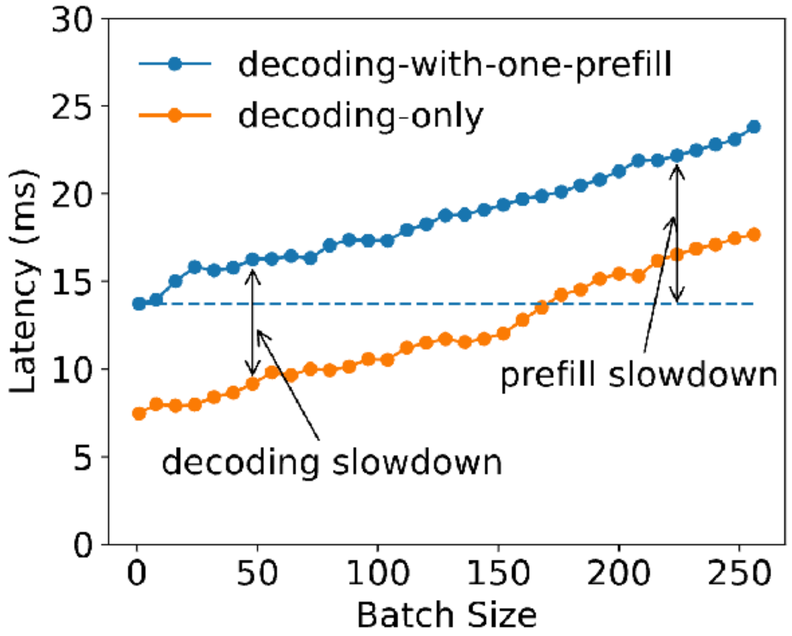
Prefill-decode disaggregation

- **Prefill**: Processes the entire sequence in parallel and store key and value vectors from the attention layers in a KV cache. Because it's handling all the tokens at once, prefill is compute-bound, but not too demanding on GPU memory.
- **Decode**: Generates the output tokens, one at a time, by reusing the KV cache built earlier. Different from prefill, decode requires fast memory access but lower compute.

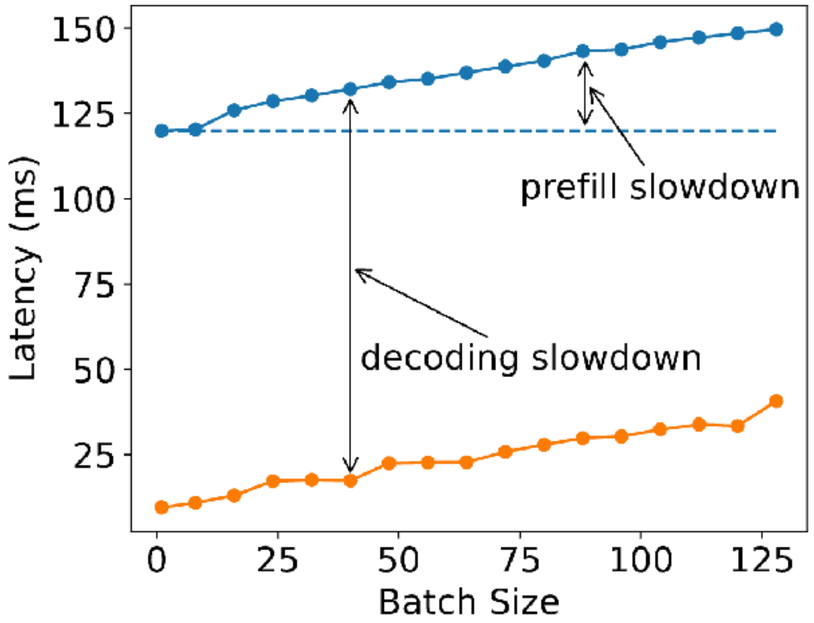


Note: Detokenization happens after each decode step. Each token (T0, T1, T2) is detokenized and output sequentially, not just the final one (T3).

Prefill-decoding interference



(a) Input length = 128



(b) Input length = 1024

Why disaggregation makes sense

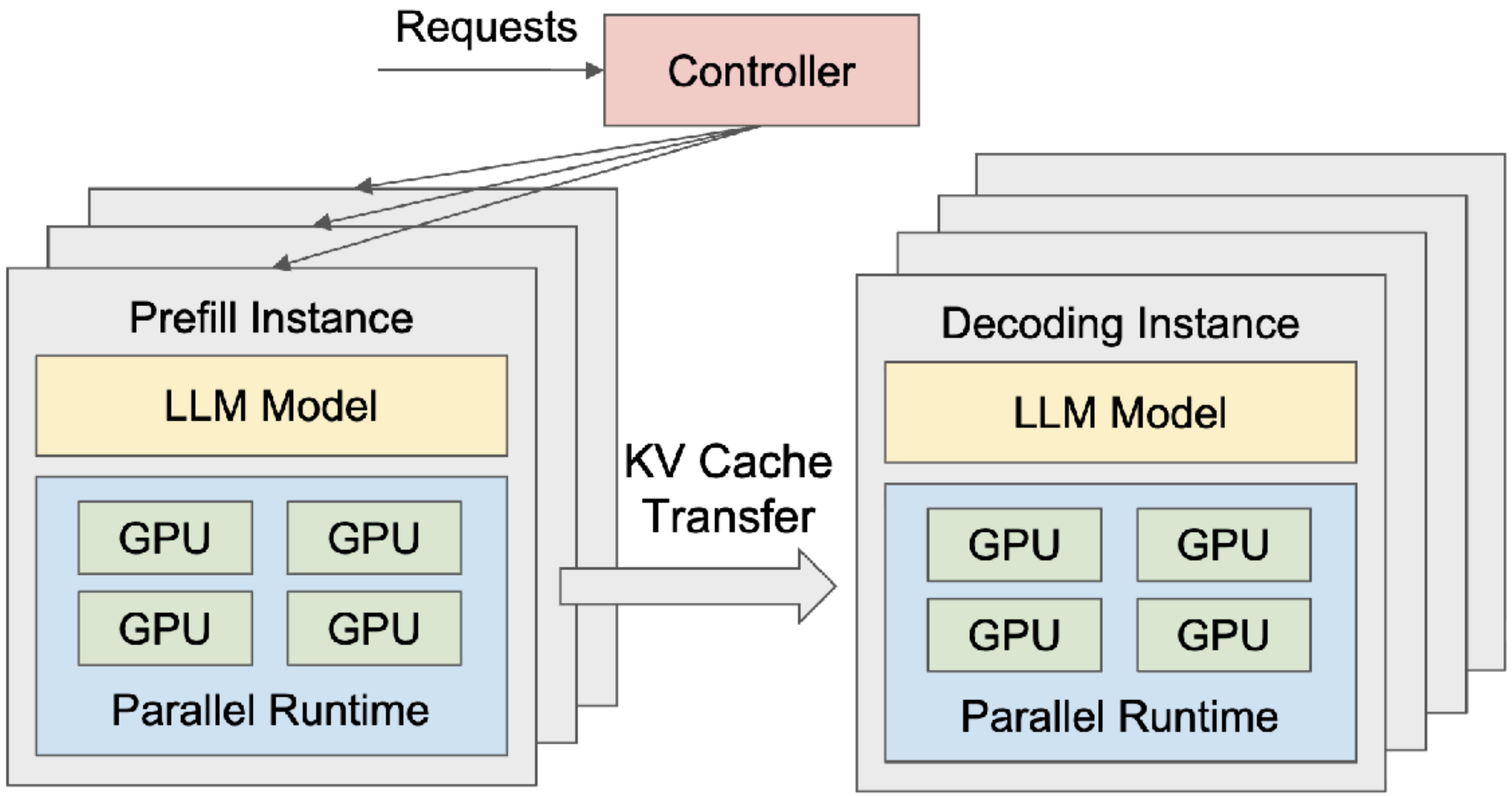
The idea of PD disaggregation is simple: separate these two very different tasks so they don’t get in each other’s way.

- **Dedicated resource allocation:** Prefill and decode can be scheduled and scaled independently on different hardware. For example, if your workload has lots of prompt overlap (like multi-turn conversations or agentic workflows), it means much of your KV cache can be reused. As a result, there’s less compute demand on prefill, and you can put more resources on decode.

- **Parallel execution:** Prefill and decode phases don’t interfere with each other anymore. You can run them more efficiently in parallel, which means better concurrency and throughput.

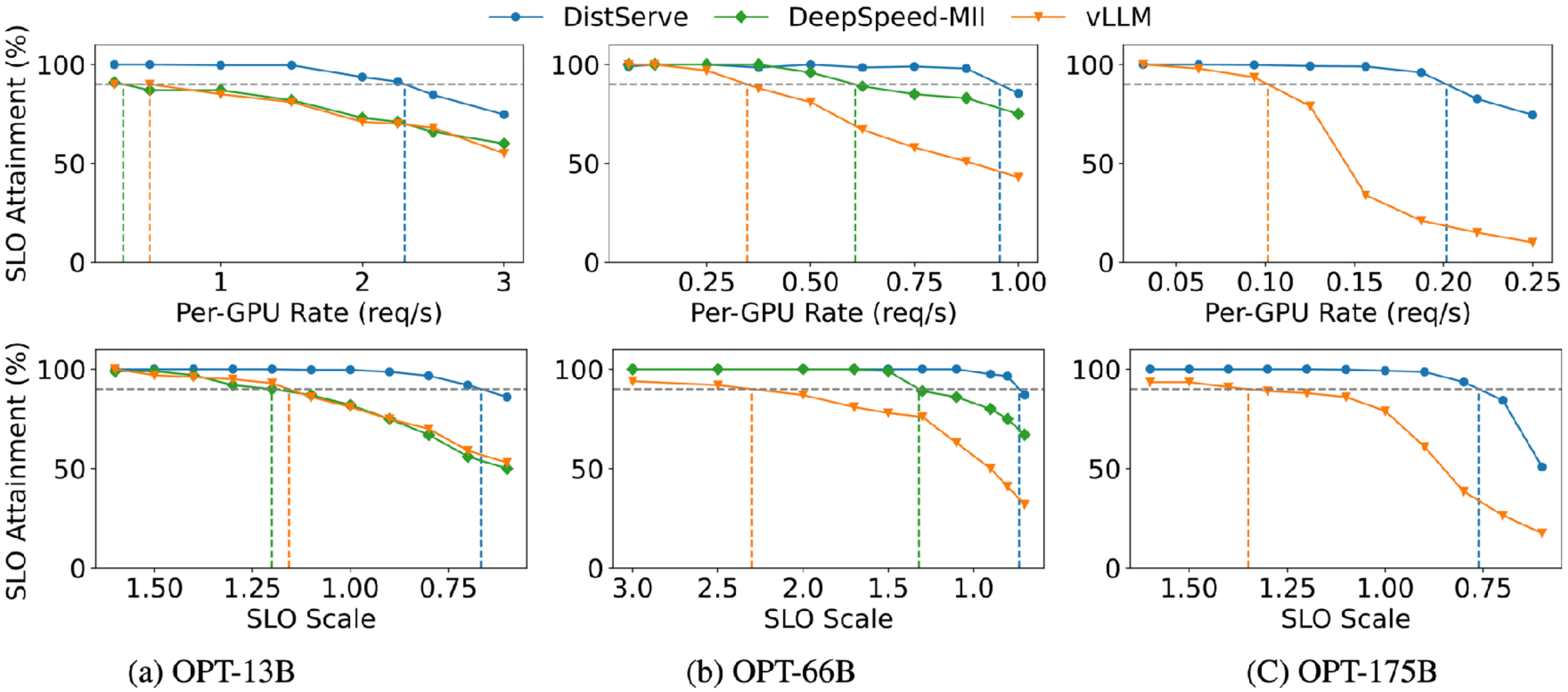
- **Independent tuning:** You can implement different optimization techniques (like tensor or pipeline parallelism) for prefill and decode to better meet your goals for TTFT and ITL.

Runtime System Architecture



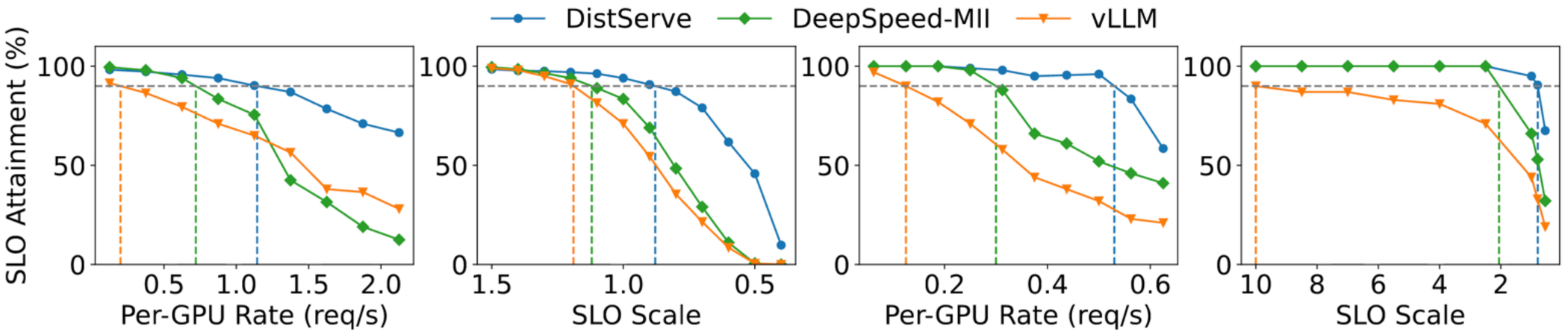
Results

Chatbot application with OPT models on the ShareGPT dataset.



Results

Code completion and summarization tasks with OPT-66B on HumanEval and LongBench datasets, respectively.

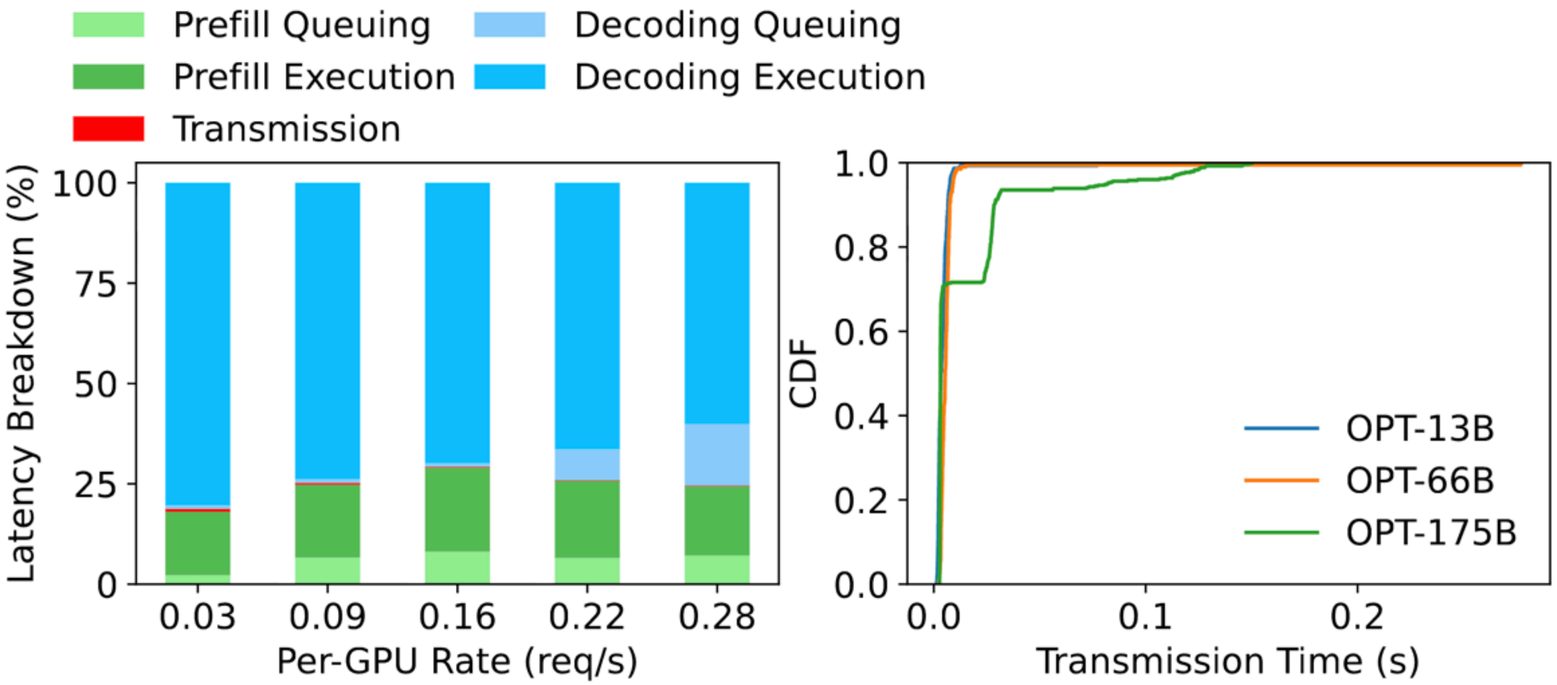


(a) Code Completion

(b) Summarization

Latency Breakdown

Left: Latency breakdown when serving OPT-175B on ShareGPT dataset with DistServe. Right: The CDF function of KV Cache transmission time for three OPT models.



Prefix caching

- Prefix caching (also known as prompt caching or context caching) is one of the most effective techniques to reduce latency and cost in LLM inference. It's especially useful in production workloads with repeated prompt structures, such as chat systems, AI agents, and RAG pipelines.

- **The idea is simple:** By caching the KV cache of an existing query, a new query that shares the same prefix can skip recomputing that part of the prompt. Instead, it directly reuses the cached results.

How does prefix caching work?

1. During prefill, the model performs a forward pass over the entire input and builds up a key-value (KV) cache for attention computation.

2. During decode, the model generates output tokens one by one, using the cached states from the prefill stage. The attention mechanism computes a matrix of token interactions. The resulting KV pairs for each token are stored in GPU memory.

3. For a new request with a matching prefix, you can skip the forward pass for the cached part and directly resume from the last token of the prefix.

Example

For example, consider a chatbot with this system prompt:

```
You are a helpful AI writer. Please write in a professional manner.
```

This prompt doesn't change from one conversation to the next. Instead of recalculating it every time, you store its KV cache once. Then, when new messages come in, you reuse this stored prefix cache, only processing the new part of the prompt.

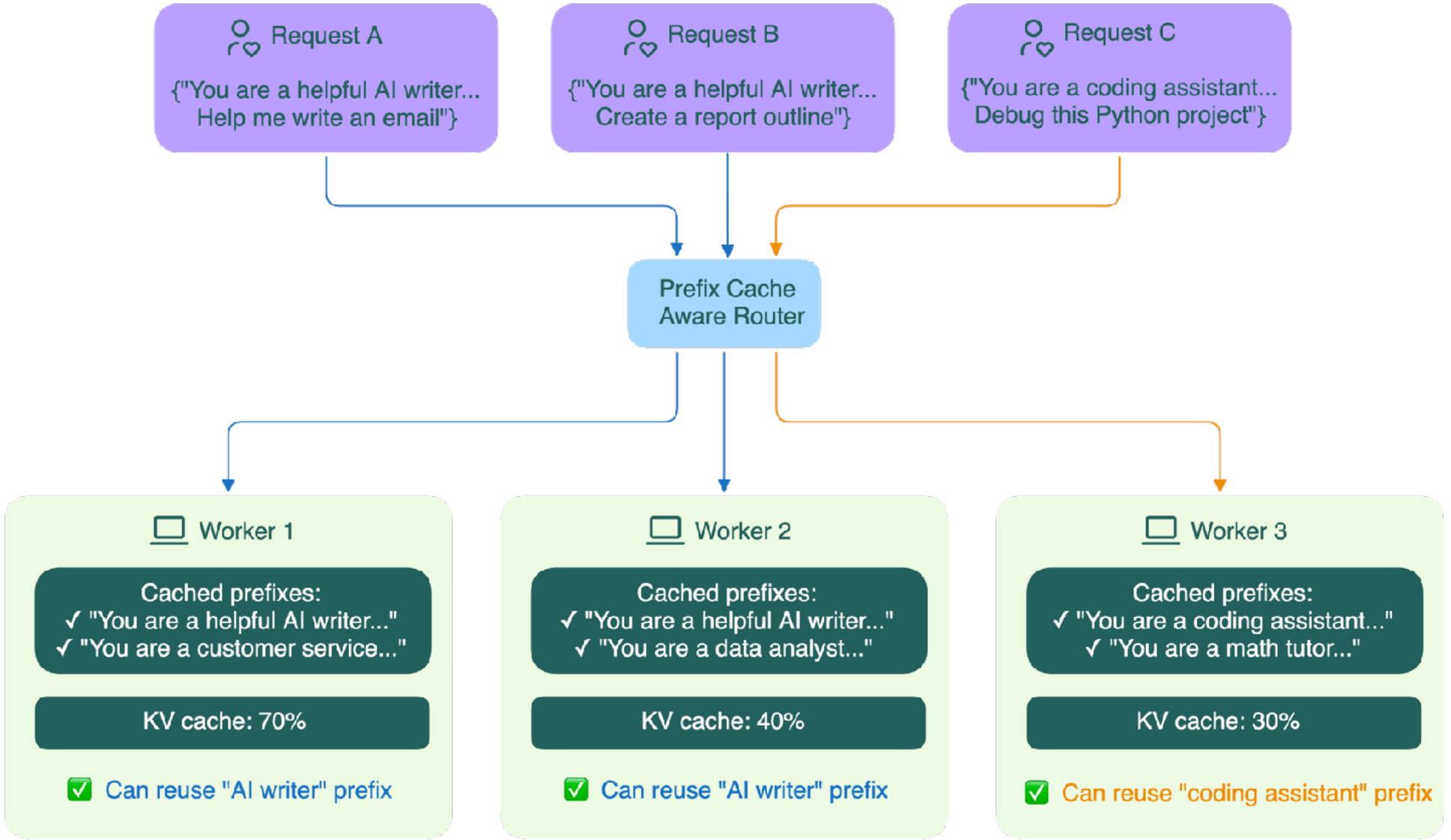
What is the difference between KV caching and prefix caching?

KV caching is used to store the intermediate attention states of each token in GPU memory. It was originally used to describe caching within a **single inference request**, especially critical for speeding up the decoding stage.

LLMs work autoregressively during decode as they output the next new token based on the previously generated tokens (i.e. reusing their KV cache). Without the KV cache, the model needs to recompute everything for the previous tokens in each decode step (and the context grows with every step), which would be a huge waste of resources.

When extending this caching concept across **multiple requests**, it's more accurate to call it prefix caching. Since the computation of the KV cache only depends on all previous tokens, different requests with identical prefixes can reuse the same cache of the prefix tokens and avoid recomputing them.

Prefix-aware routing



KV cache utilization-aware load balancing