11868 LLM Systems
LLM Serving

Lei Li

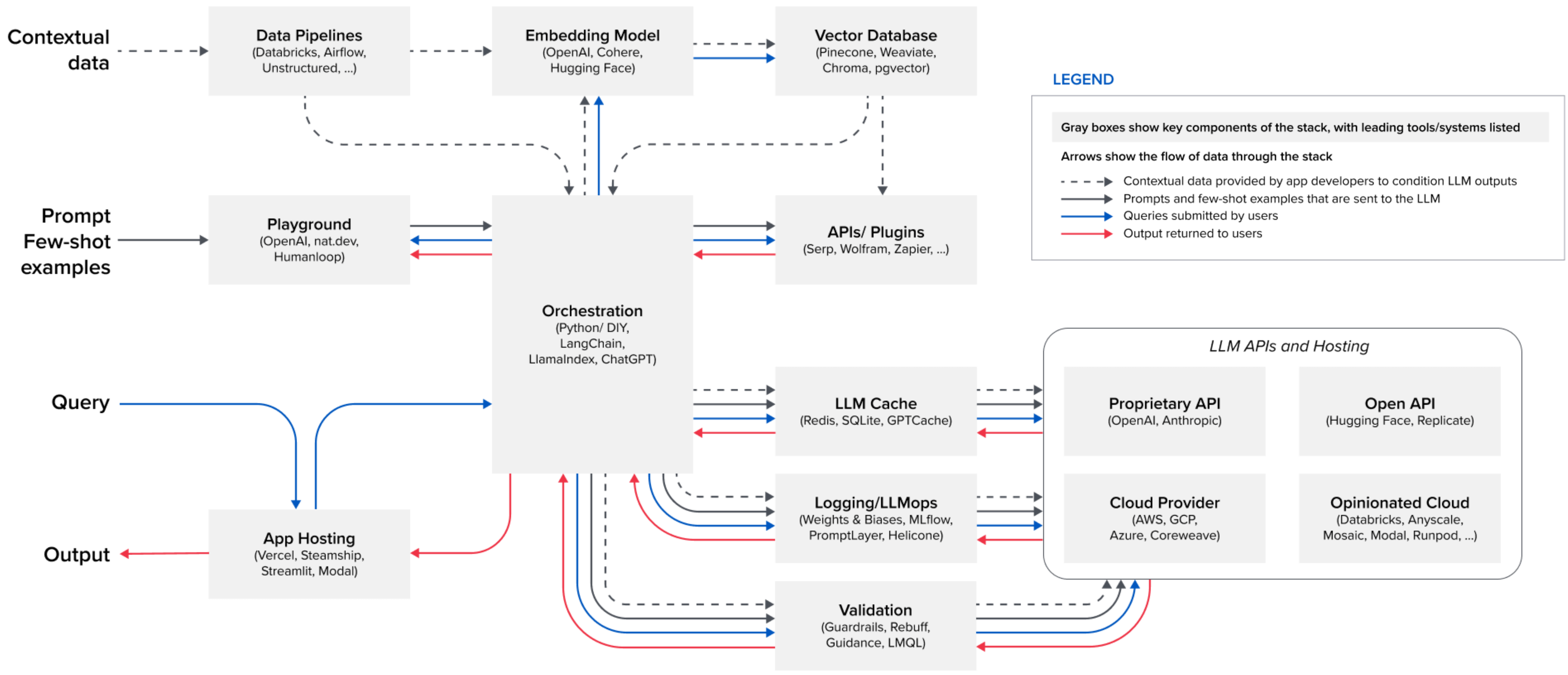
Carnegie Mellon University
Language Technologies Institute

Today’s Topic

1. Architectures for LLM Applications

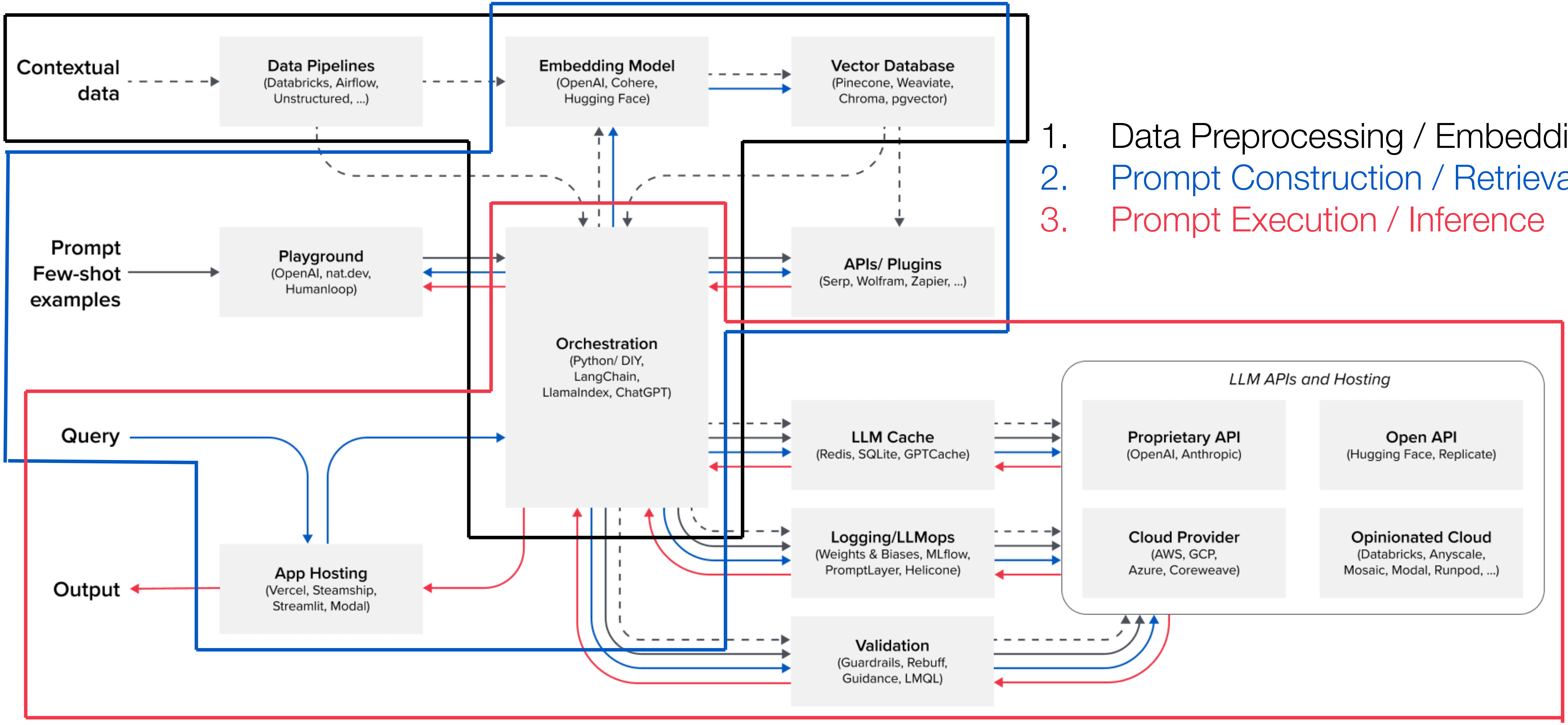
2. Frameworks for LLM Serving

LLM App Stack



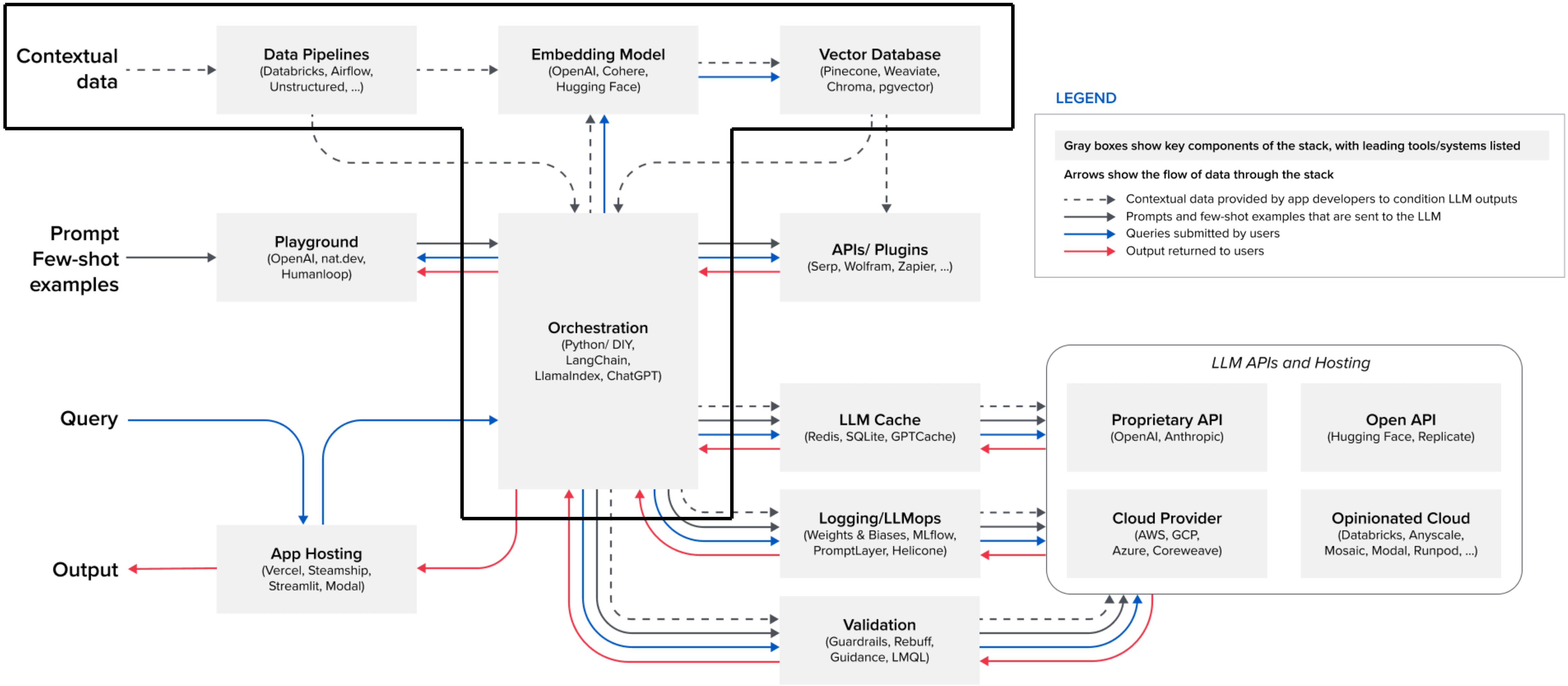
https://a16z.com/emerging-architectures-for-llm-applications/

LLM App Stack



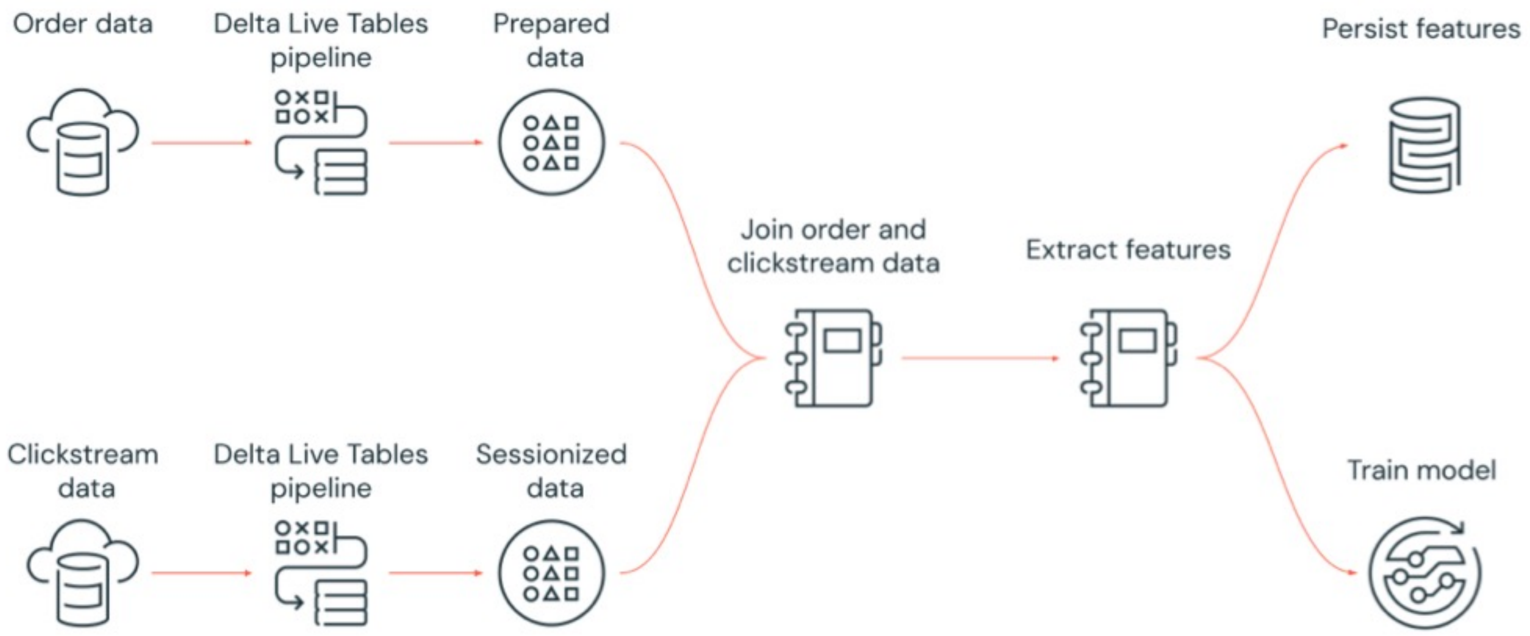
1. Data Preprocessing / Embedding
2. Prompt Construction / Retrieval
3. Prompt Execution / Inference

1. Data Preprocessing / Embedding



1. Data Preprocessing / Embedding

- Data Pipelines
  - Include loaders, parsers, and preprocessing.
  - E.g. Databricks, Airflow, and Unstructured.



1. Data Preprocessing / Embedding

- Embedding Model
    - Different models have their own advantages.
    - OpenAI: Effective but cheap
    - Cohere: Focus more on embeddings
    - Hugging Face: Open-source
    - Customer Embedding: e.g. BERT

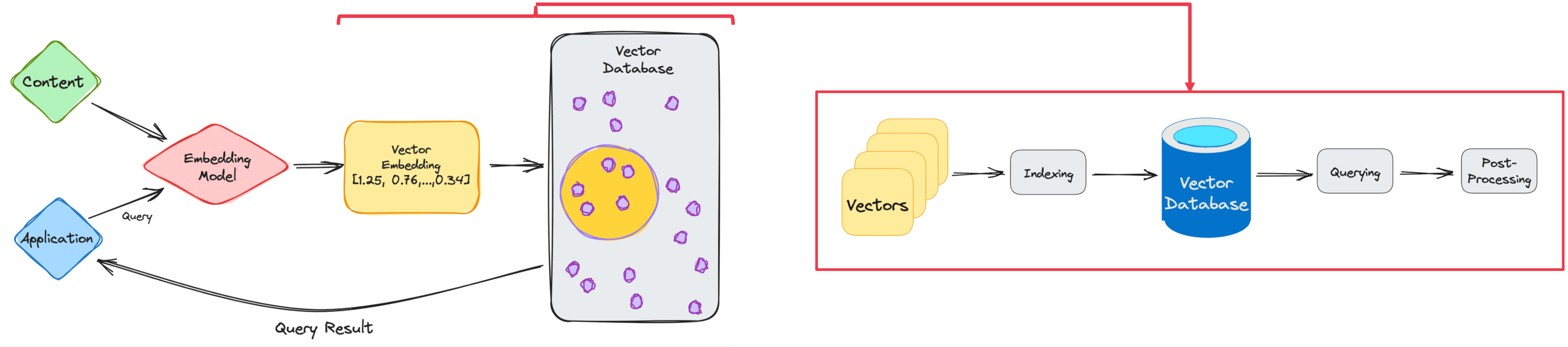
OpenAI

cohere

Hugging Face

1. Data Preprocessing / Embedding

- Vector Database
  - Offer optimized storage and query capabilities for the unique structure of vector embeddings.



1. Data Preprocessing / Embedding

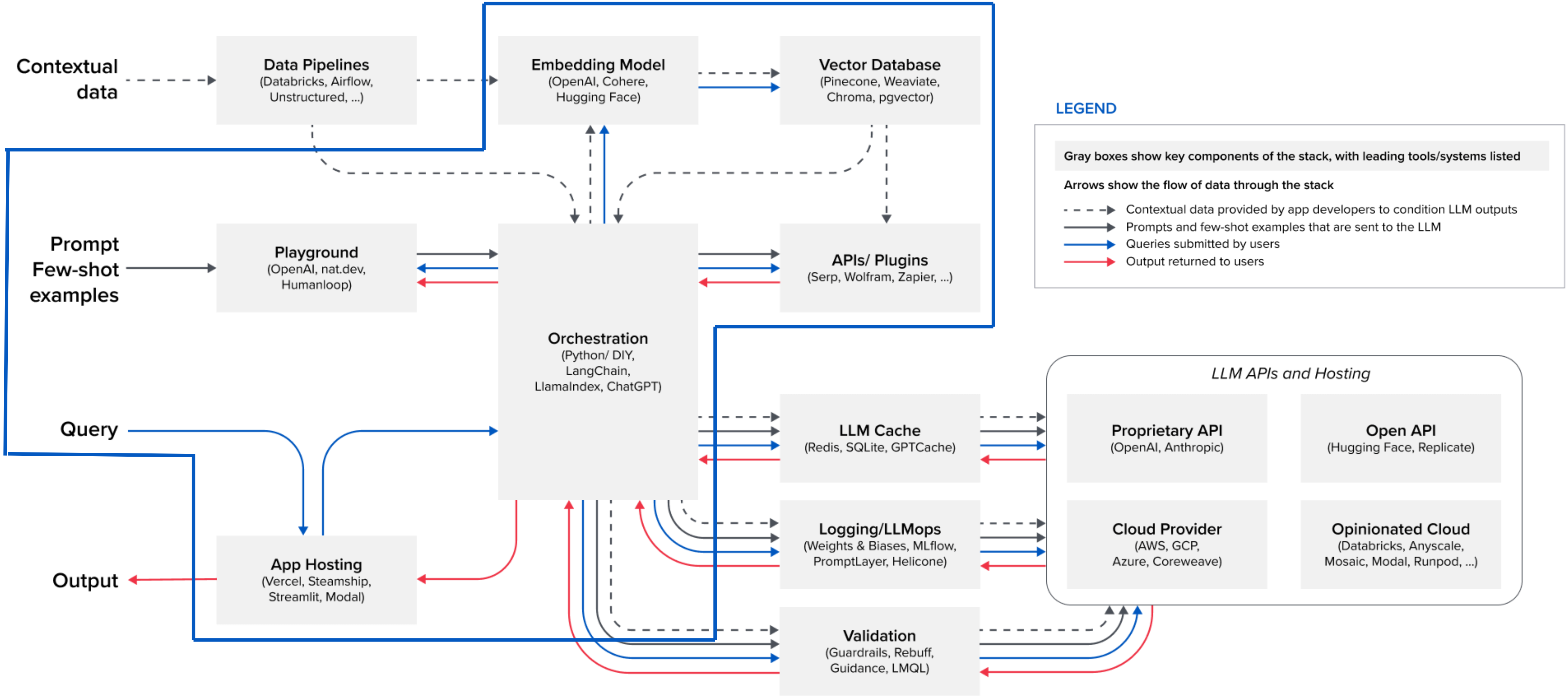
- Vector Database
  - Pinecone: Fully clound-hosted
  - chroma: Local vector management
  - Faiss: by meta
  - Weaviate: Open-source

Pinecone

chroma

Weaviate

2. Prompt Construction / Retrieval

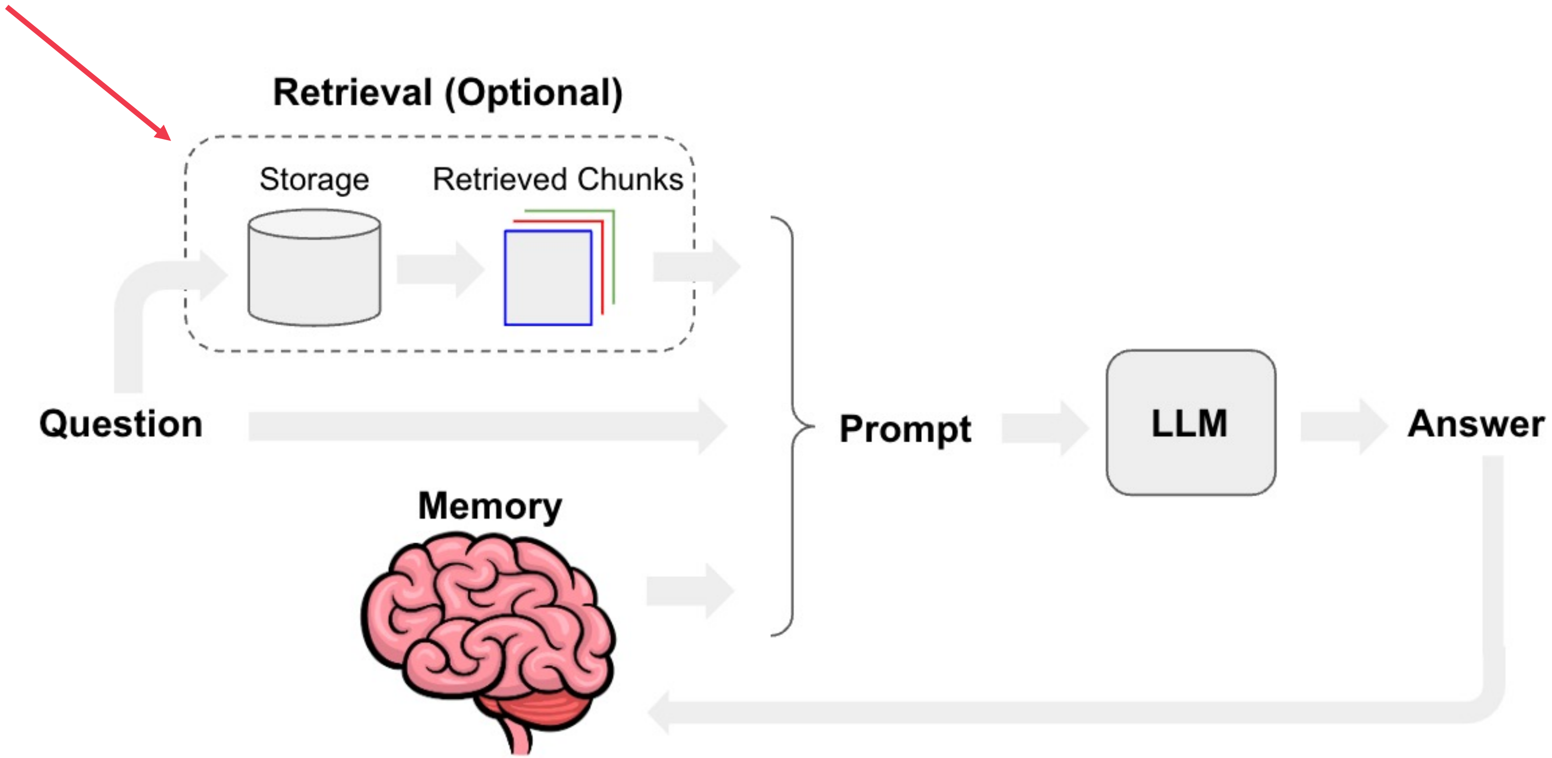


2. Prompt Construction / Retrieval

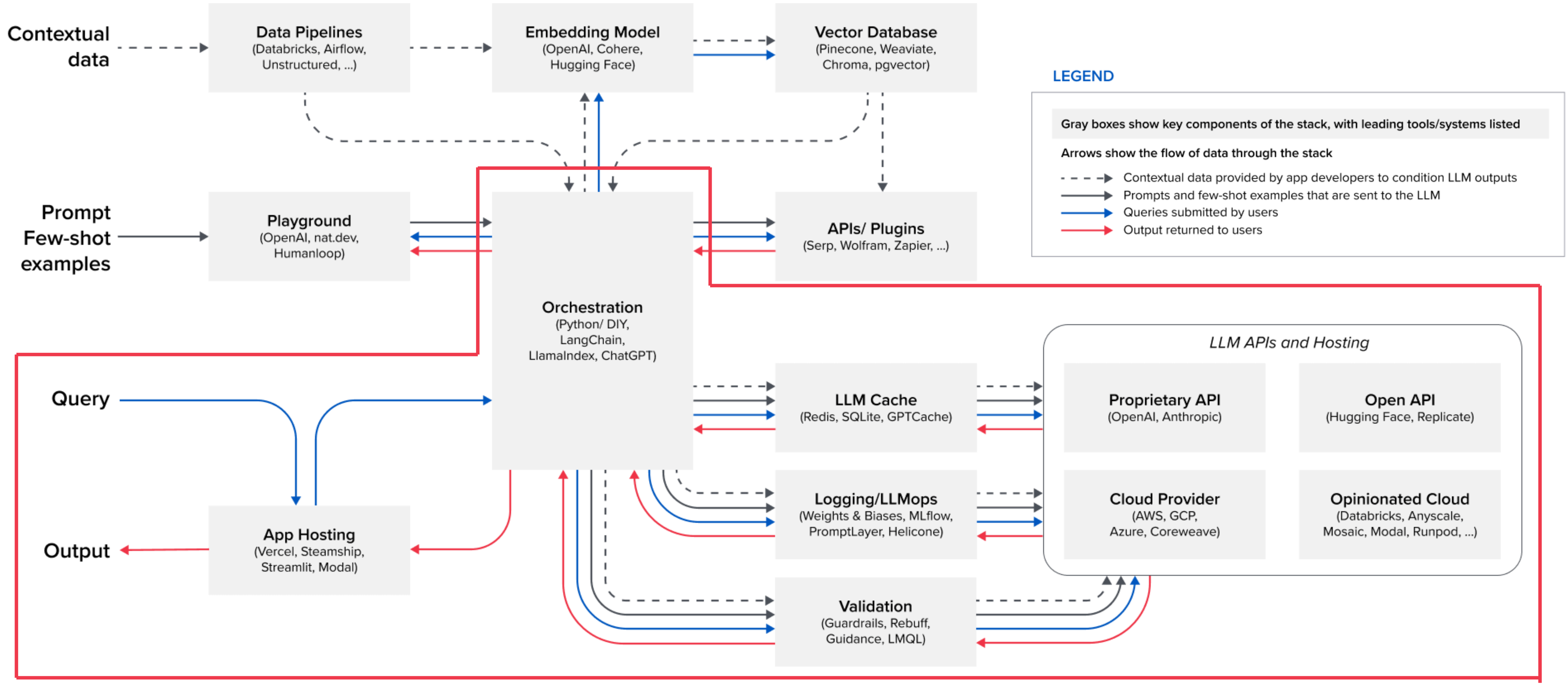
- Orchestration (Prompt Optimization/Engineering)
    - Abstract away many of the details of prompt chaining
    - Interface with external APIs
    - Retrieve contextual data from vector databases
    - Maintain memory across multiple LLM calls
    - Output a prompt, or a series of prompts, to submit to an LLM

2. Prompt Construction / Retrieval

- Orchestration
  - LlamaIndex: Specially designed for building search and retrieval applications
  - LangChain: A general-purpose framework for a wide variety of applications



3. Prompt Execution / Inference



3. Prompt Execution / Inference

- LLM APIs
  - Proprietary APIs: E.g. OpenAI and Anthropic.
  - Open APIs: E.g. Hugging Face and Replicate.

  - ChatGPT: May have unreliable latency for production.

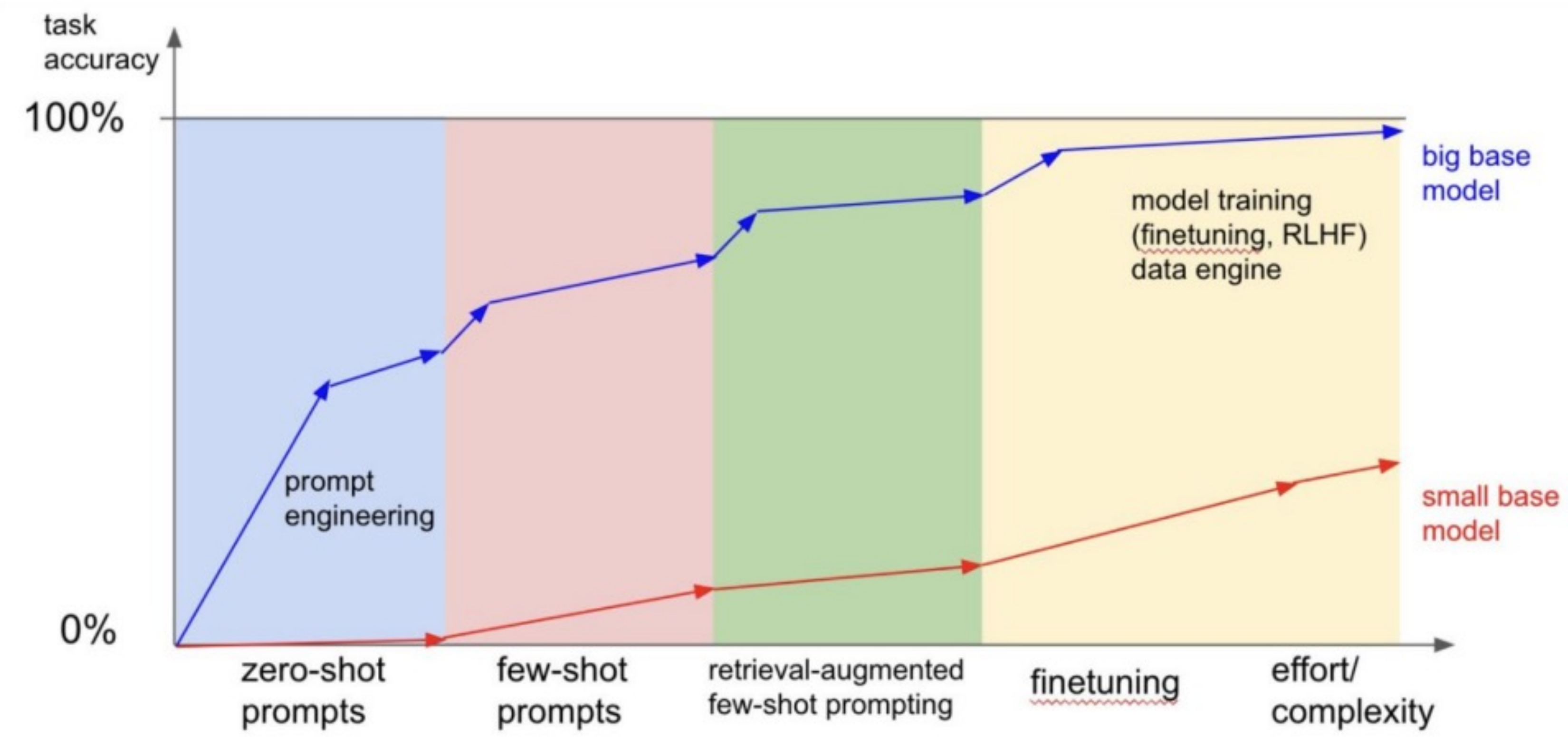
# tokens	p50 latency (sec)	p75 latency	p90 latency
input: 51 tokens, output: 1 token	0.58	0.63	0.75
input: 232 tokens, output: 1 token	0.53	0.58	0.64
input: 228 tokens, output: 26 tokens	1.43	1.49	1.62

3. Prompt Execution / Inference

- LLM Hosting
  - General Could : E.g. AWS, GCP, and Azure.
  - Opinionated Cloud: E.g. Databricks, Anyscale, and Mosaic.
- LLM Cache
  - E.g. Redis, SQLite, and GPTCache.
- Validation
  - E.g. Guardrails, Rebuff, and Guidance.

3. Prompt Execution / Inference

- Logging
  - E.g. Weights&Biases, MLflow, and PromptLayer.

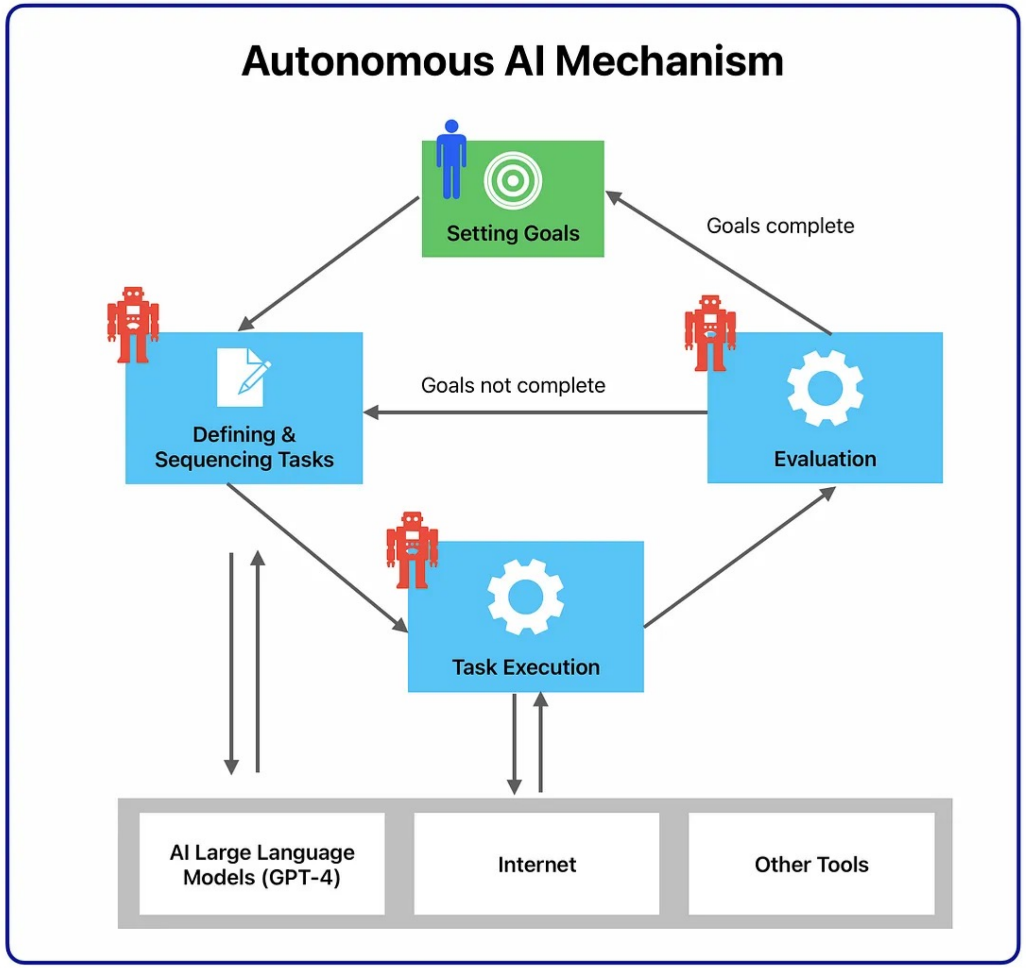


3. Prompt Execution / Inference

- App Hosting
  - Vercel: Provide a cloud-native solution
  - Steamship: end-to-end hosting
  - Anyscale and Modal: Host models and Python code in one place

Build App Autonomously with AI Agent

- Can we use AI to perform any task?
- AutoGPT
  - Let an LLM-based AI agent decide what to do, while feeding its results back into the prompt.
  - This allows the program to iteratively and incrementally work towards its objective.



Today’s Topic

1. Architectures for LLM Applications

2. Frameworks for LLM Serving

Frameworks for LLM Serving

- In this lecture:
  - NVIDIA Triton + LightSeq/TensorRT-LLM
  - Text Generation Inference
  - OpenLLM
  - MLC LLM
  - LightLLM
- In the later lectures:
  - vLLM / DeepSpeed

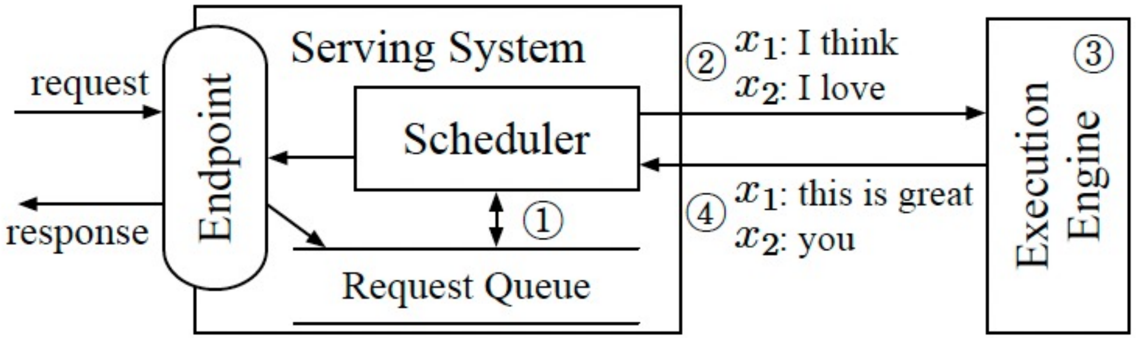


Figure 2: Overall workflow of serving a generative language model with existing systems.

Triton“s”

- Triton (Inference Server) from NVIDIA
  - An open-source inference serving software that streamlines AI inferencing.
- Triton from OpenAI
  - A language and compiler for writing highly efficient custom Deep-Learning primitives.

NVIDIA Triton

- It supports for various deep learning frameworks.
  - E.g. TensorFlow and PyTorch.
- It optimizes inference for multiple query types.
  - E.g. real-time, batch, and streaming.
- It can be deployed on different environments
  - E.g. public cloud and embedded devices.

NVIDIA Triton – Dynamic Batching

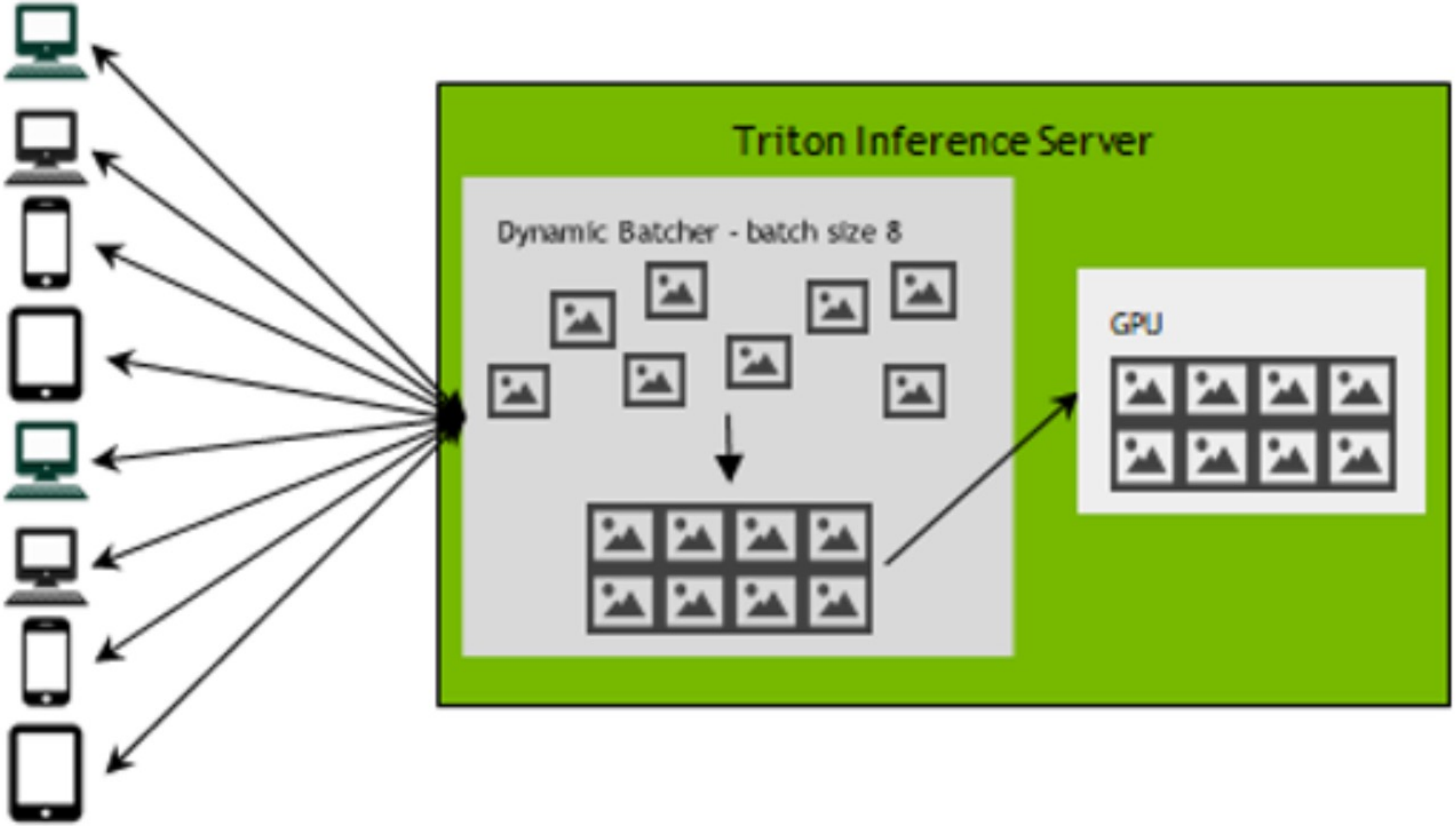


Figure 2. NVIDIA Triton dynamic batching

NVIDIA Triton + LightSeq / TensorRT-LLM

- For LLM serving, Triton has to use with an inference engine, e.g., LightSeq or TensorRT-LLM.
- Triton groups multiple client requests into a batch.
- TensorRT-LLM conducts the inference procedure in the batched manner.

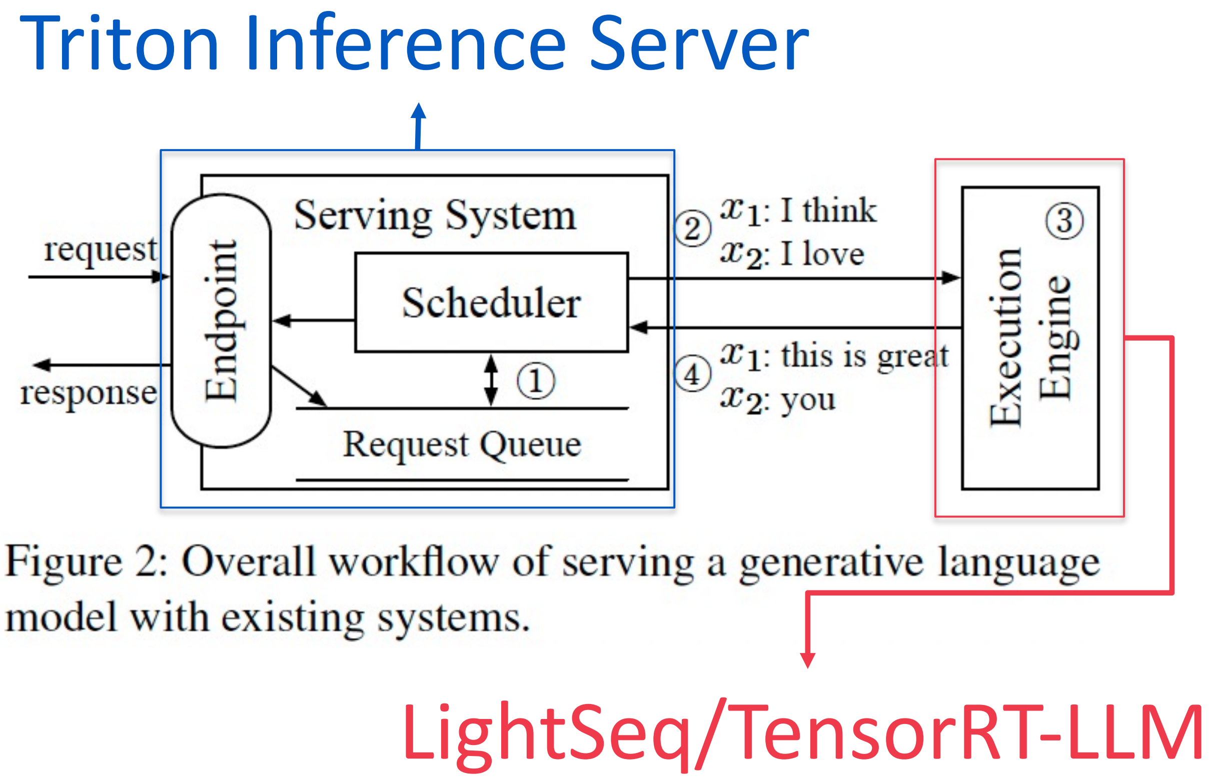


Figure 2: Overall workflow of serving a generative language model with existing systems.

NVIDIA Triton + FasterTransformer – Code

Run web server using docker:

```
# Step 1: Clone fastertransformer_backend from the Triton GitHub repository
git clone https://github.com/triton-inference-server/fastertransformer_backend.git
cd fastertransformer_backend && git checkout -b t5_gptj_blog remotes/origin/dev/t5_gptj_blog

# Step 2: Build Docker container with Triton and FasterTransformer libraries
docker build --rm  --build-arg TRITON_VERSION=22.03 -t triton_with_ft:22.03 \
             -f docker/Dockerfile .
cd ../
docker run -it --rm --gpus=all --shm-size=4G  -v $(pwd):/ft_workspace \
           -p 8888:8888 triton_with_ft:22.03 bash
apt install jupyter-lab && jupyter lab -ip 0.0.0.0

# Steps 3 and 4: Clone FasterTransformer source codes and build the library
git clone https://github.com/NVIDIA/FasterTransformer.git
mkdir -p FasterTransformer/build && cd FasterTransformer/build
git submodule init && git submodule update
cmake -DSM=xx -DCMAKE_BUILD_TYPE=Release -DBUILD_PYT=ON -DBUILD_MULTI_GPU=ON ..
make -j32

# Step 5 (GPT-J): Download and prepare weights of the GPT-J model
wget https://mystic.the-eye.eu/public/AI/GPT-J-6B/step_383500_slim.tar.zstd
tar -axf step_383500_slim.tar.zstd -C ./models/

# Step 6 (GPT-J): Convert weights into FT format
python3 ./FasterTransformer/examples/pytorch/gptj/utils/gptj_ckpt_convert.py \
          --output-dir ./models/j6b_ckpt \
          --ckpt-dir ./step_383500/ \
          --n-inference-gpus 2

# Step 7 (GPT-J): Kernel-autotuning for the GPT-J inference
./FasterTransformer/build/bin/gpt_gemm 8 1 32 12 128 6144 51200 1 2

# Step 8 (GPT-J): Prepare the Triton config and serve the model
CUDA_VISIBLE_DEVICES=0,1 /opt/tritonserver/bin/tritonserver  --model-repository=./triton-model-store/gptj/ &
```

NVIDIA Triton + FasterTransformer – Code

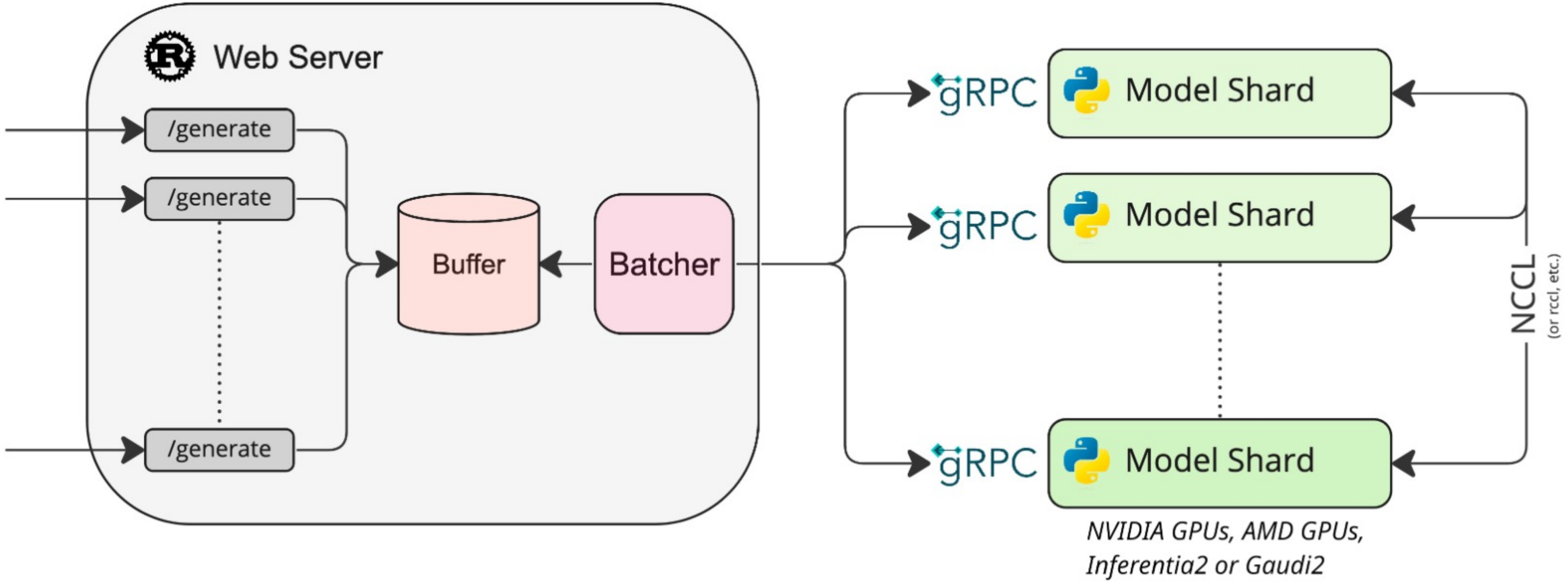
Make queries:

```python
# Import libraries
import tritonclient.http as httpclient

# Initizlize client
client = httpclient.InferenceServerClient("localhost:8000",
                                          concurrency=1,
                                          verbose=False)
# ...

# Sending request
result = client.infer(MODEl_GPTJ_FASTERTRANSFORMER, "Funniest joke ever:")
print(result.as_numpy("OUTPUT_0"))
```

Text Generation Inference

- It has optimized transformers code for inference using Flash Attention and Paged Attention.
    - Native support for models from HuggingFace.
- It is production ready and can monitor the server load and get insights into its performance.
    - Distributed tracing with Open Telemetry, Prometheus metrics.
- It offers a wide range of options to manage model inference.
    - E.g. precision adjustment and quantization.

Text Generation Inference – Framework



tensor-parallel

Text Generation Inference – Code

Run web server using docker:

```
mkdir data
docker run --gpus all --shm-size 1g -p 8080:80 \
-v data:/data ghcr.io/huggingface/text-generation-inference:0.9 \
  --model-id huggyllama/llama-13b \
  --num-shard 1
```

Make queries:

```
# pip install text-generation
from text_generation import Client

client = Client("http://127.0.0.1:8080")
prompt = "Funniest joke ever:"
print(client.generate(prompt, max_new_tokens=17 temperature=0.95).generated_text)
```

OpenLLM

- It allows users to easily create AI applications by composing LLMs with other models and services.
  - E.g. LangChain, LlamaIndex, and Hugging Face.

- It allows users to bring their own LLM and fine-tune them.

- It is an open-source platform and constantly developing.

OpenLLM – Code

Run web server:

```
pip install openllm scipy
openllm start llama --model-id huggyllama/llama-13b \
  --max-new-tokens 200 \
  --temperature 0.95 \
  --api-workers 1 \
  --workers-per-resource 1
```

Make queries:

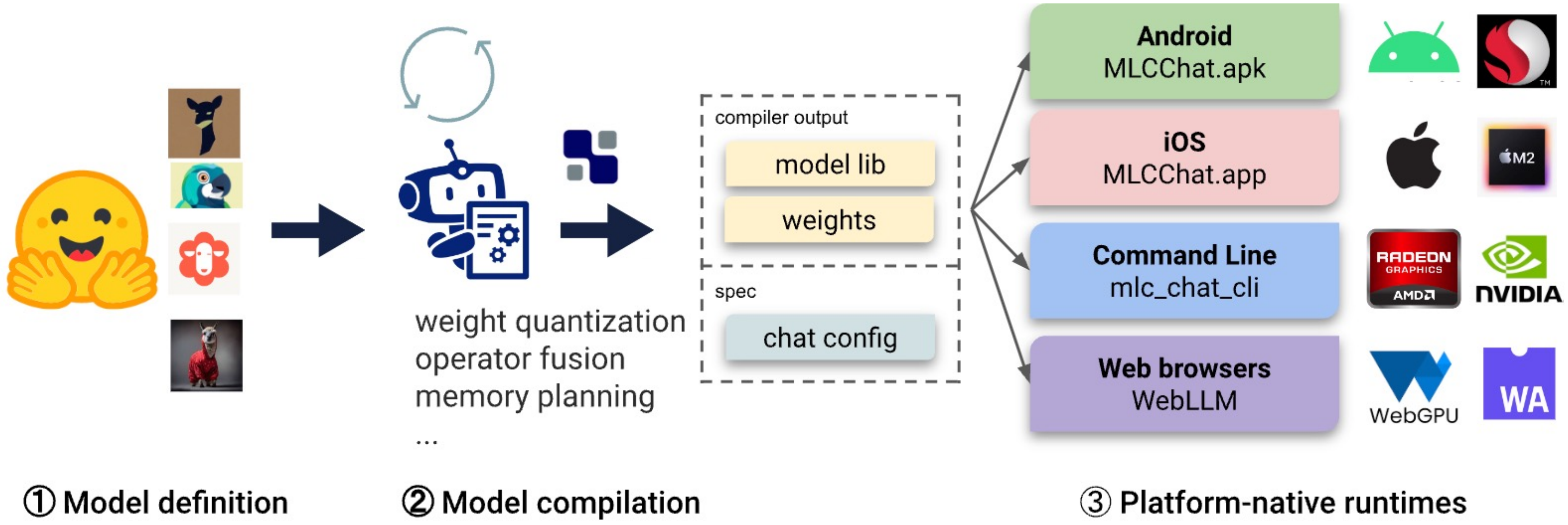
```python
import openllm

client = openllm.client.HTTPClient('http://localhost:3000')
print(client.query("Funniest joke ever:"))
```

MLC LLM

- It enables the users to develop, optimize and deploy AI models natively on consumer devices.

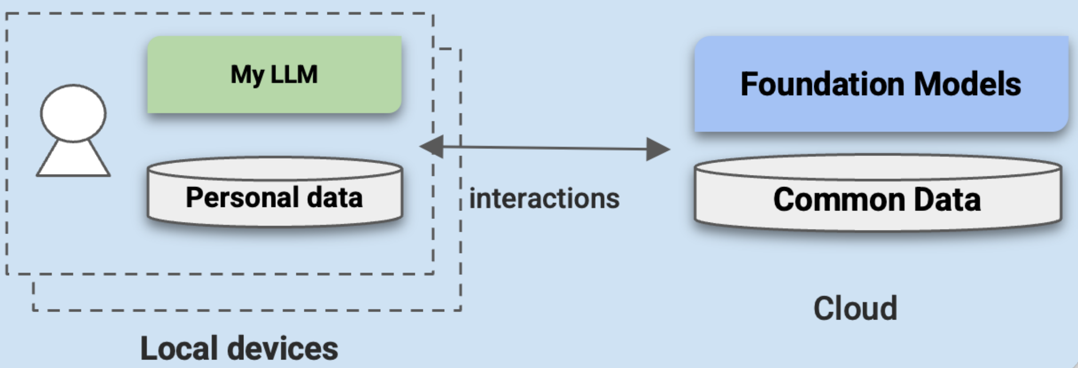
- It can compile the model for different platforms.
  - E.g. C++ for the command line, JavaScript for the web, Swift for iOS, and Java/Kotlin for Android.

MLC LLM – Framework

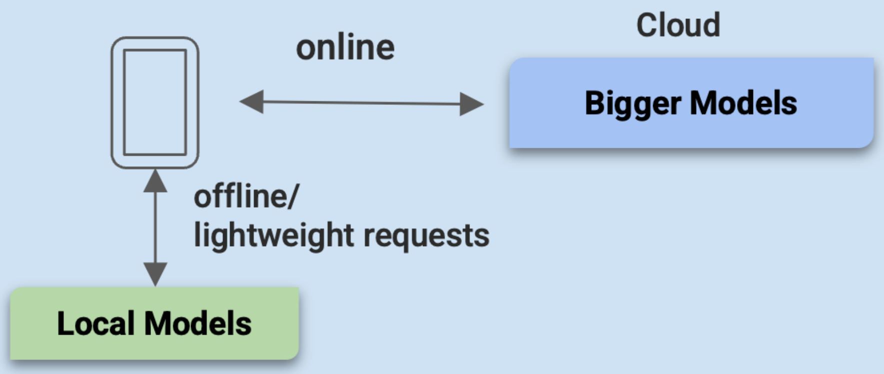


MLC LLM – Advantage of Local LLM

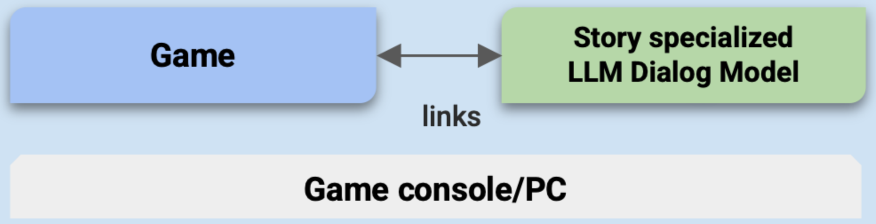
Personalization



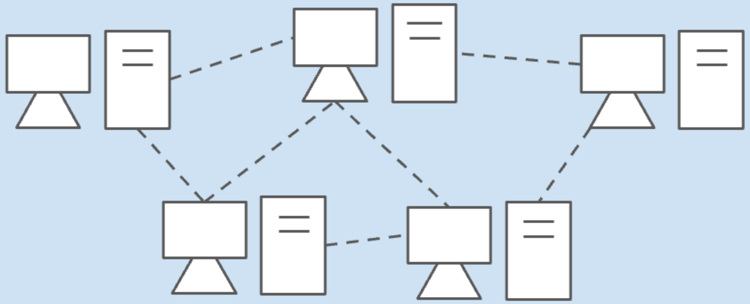
Offline Support



Specialization and App Embedding



Decentralization



MLC LLM – Code

Run web server:

```
# 1. Make sure that you have python >= 3.9
# 2. You have to run it using conda:
conda create -n mlc-chat-venv -c mlc-ai -c conda-forge mlc-chat-nightly
conda activate mlc-chat-venv

# 3. Then install package:
pip install --pre --force-reinstall mlc-ai-nightly-cu118 \
  mlc-chat-nightly-cu118 \
  -f https://mlc.ai/wheels

# 4. Download the model weights from HuggingFace and binary libraries:
git lfs install && mkdir -p dist/prebuilt && \
  git clone https://github.com/mlc-ai/binary-mlc-llm-libs.git dist/prebuilt/lib && \
  cd dist/prebuilt && \
  git clone https://huggingface.co/huggyllama/llama-13b dist/ && \
  cd ../..


# 5. Run server:
python -m mlc_chat.rest --device-name cuda --artifact-path dist
```

Make queries:

```
import requests

payload = {
   "model": "lama-30b",
   "messages": [{"role": "user", "content": "Funniest joke ever:"}],
   "stream": False
}
r = requests.post("http://127.0.0.1:8000/v1/chat/completions", json=payload)
print(r.json()['choices'][0]['message']['content'])
```

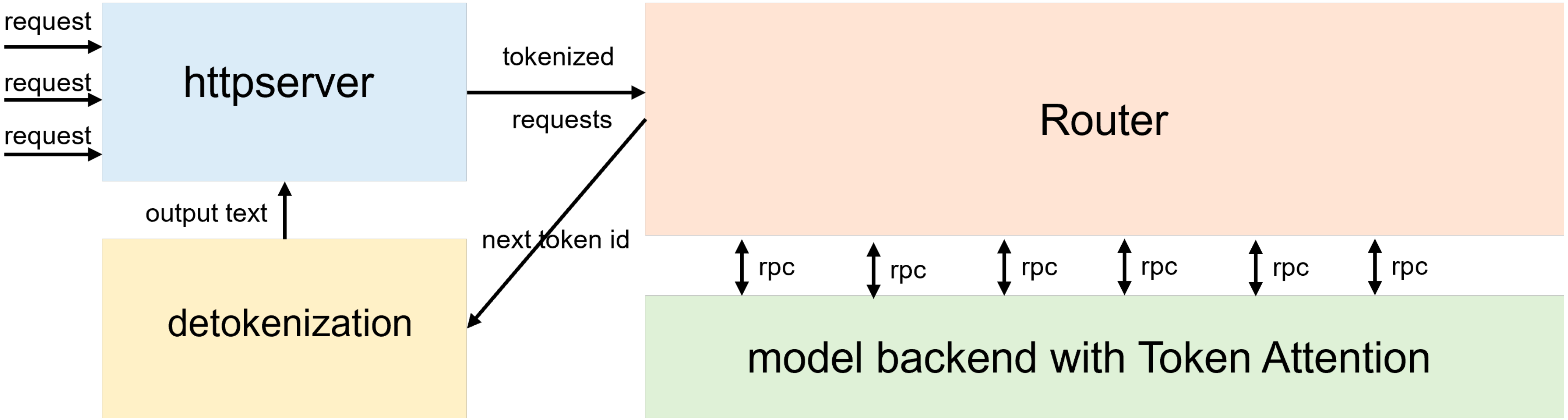
LightLLM

- It performs tokenization, model inference, and detokenization asynchronously, to improve GPU utilization.
- It implements Token Attention, a token-wise's KV cache memory management algorithm.
- It adopts Efficient Router scheduling implementation, which collaborates with Token Attention to manage the GPU memory of each token.

LightLLM – Framework

LIGHTLLM

A Light and Fast Inference Service for LLM



LightLLM – Token Attention

1. Init the Memory Cache

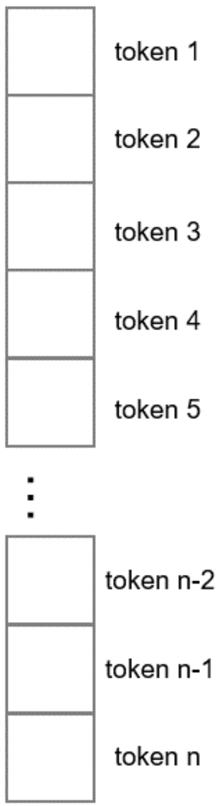
Seq1: The capital of China is

Seq2: Shanghai is

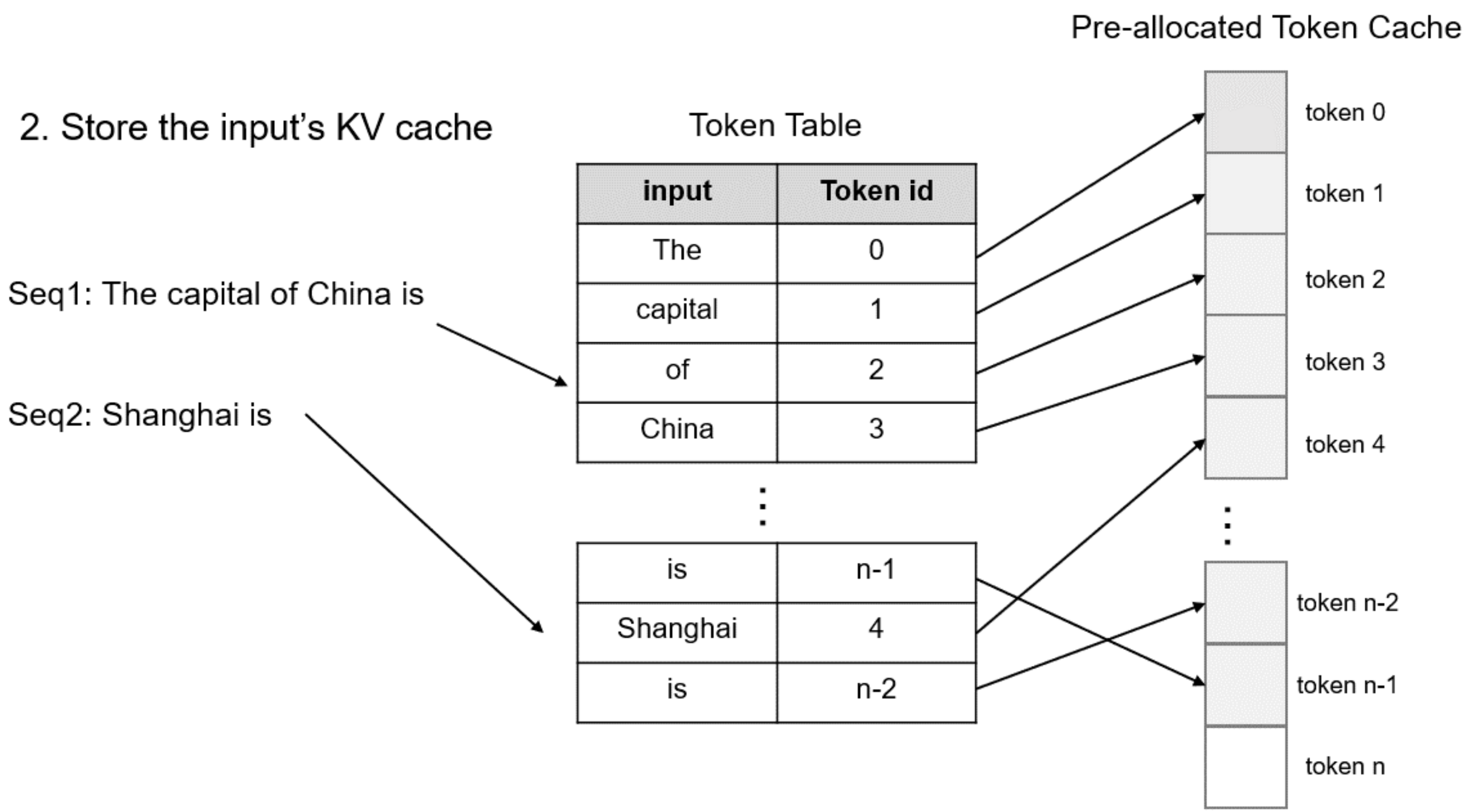
Token Table

Input	Token id
-	-
-	-
-	-
-	-
⋮	
-	-
-	-

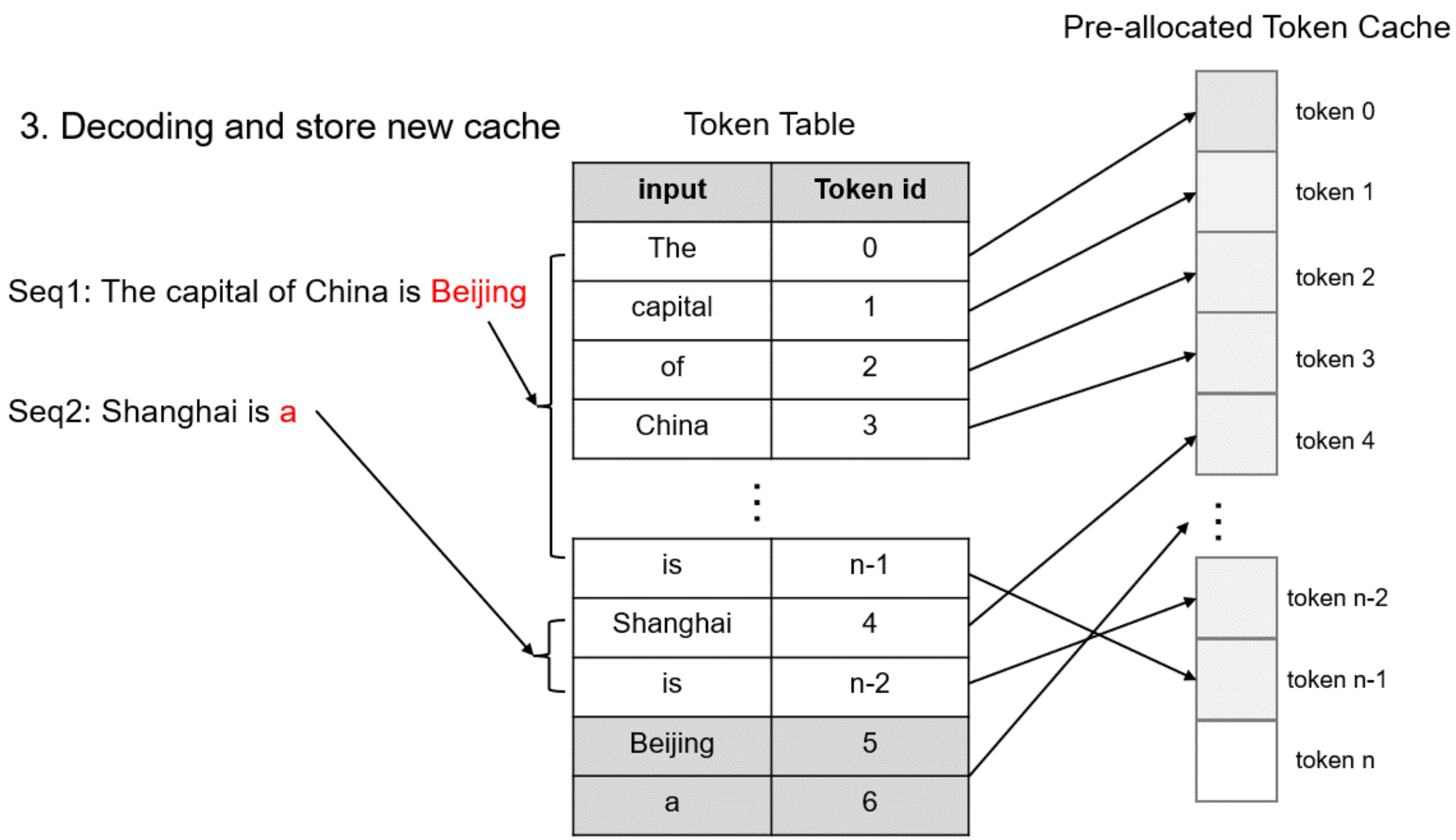
Pre-allocated Token Cache



LightLLM – Token Attention



LightLLM – Token Attention



LightLLM – Token Attention

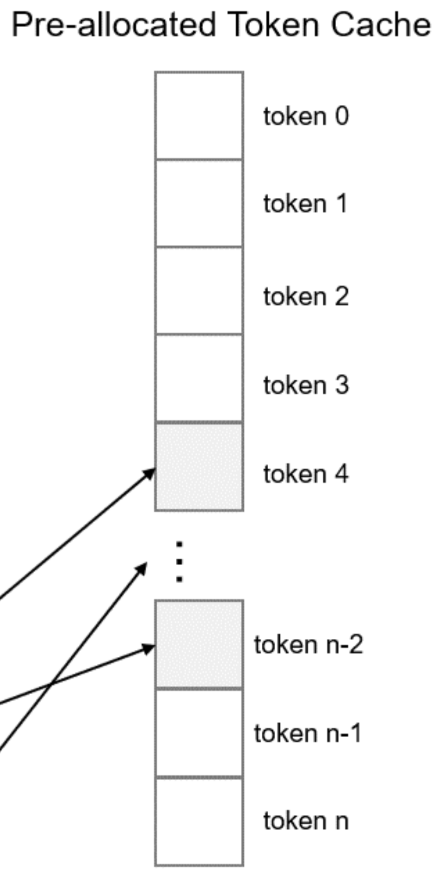
4. Free finished input Seq 1

Seq1: The capital of China is Beijing

Seq2: Shanghai is a

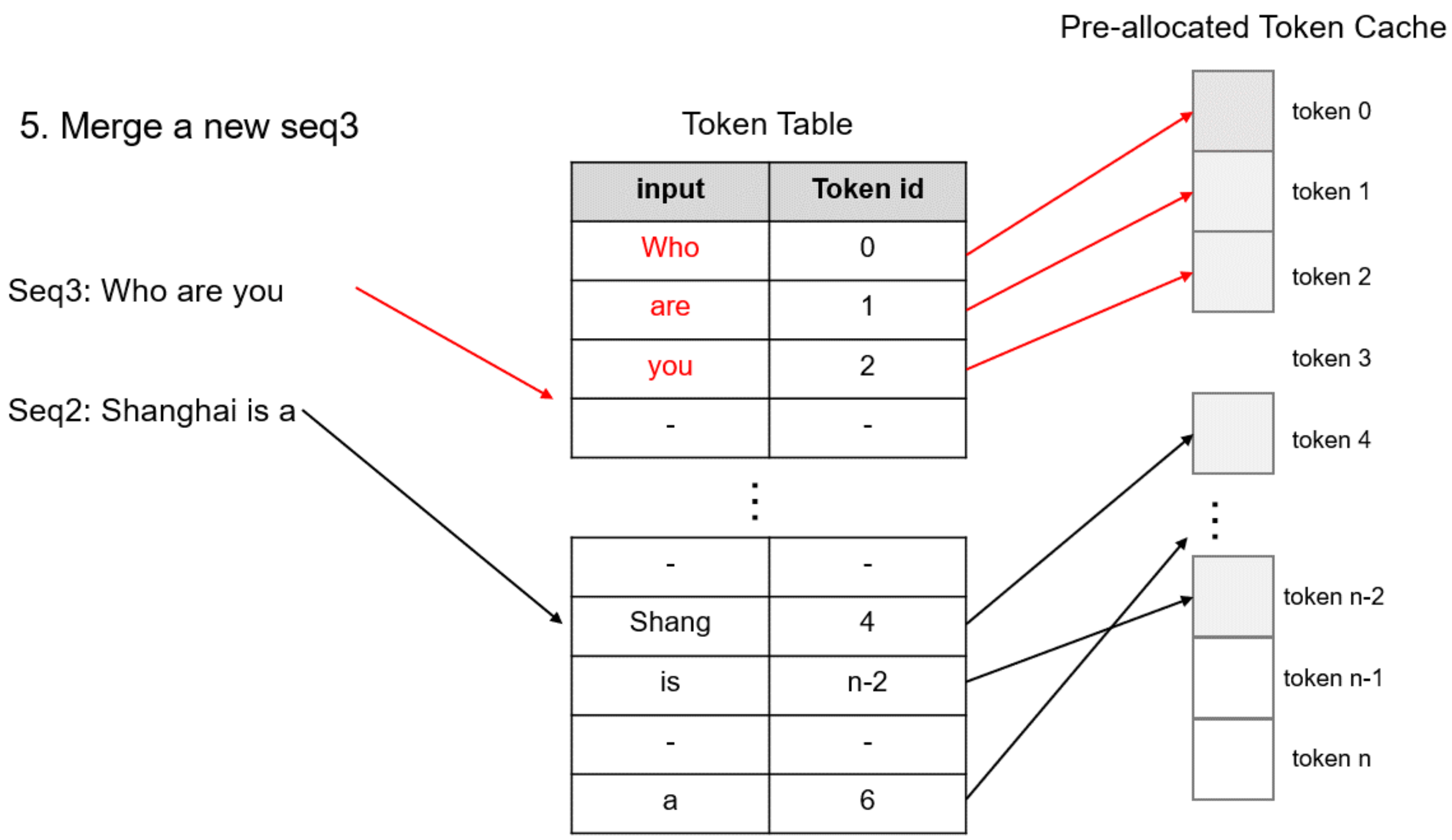
Token Table

input	Token id
-	-
-	-
-	-
-	-
⋮	
-	-
Shang	4
is	n-2
-	-
a	6

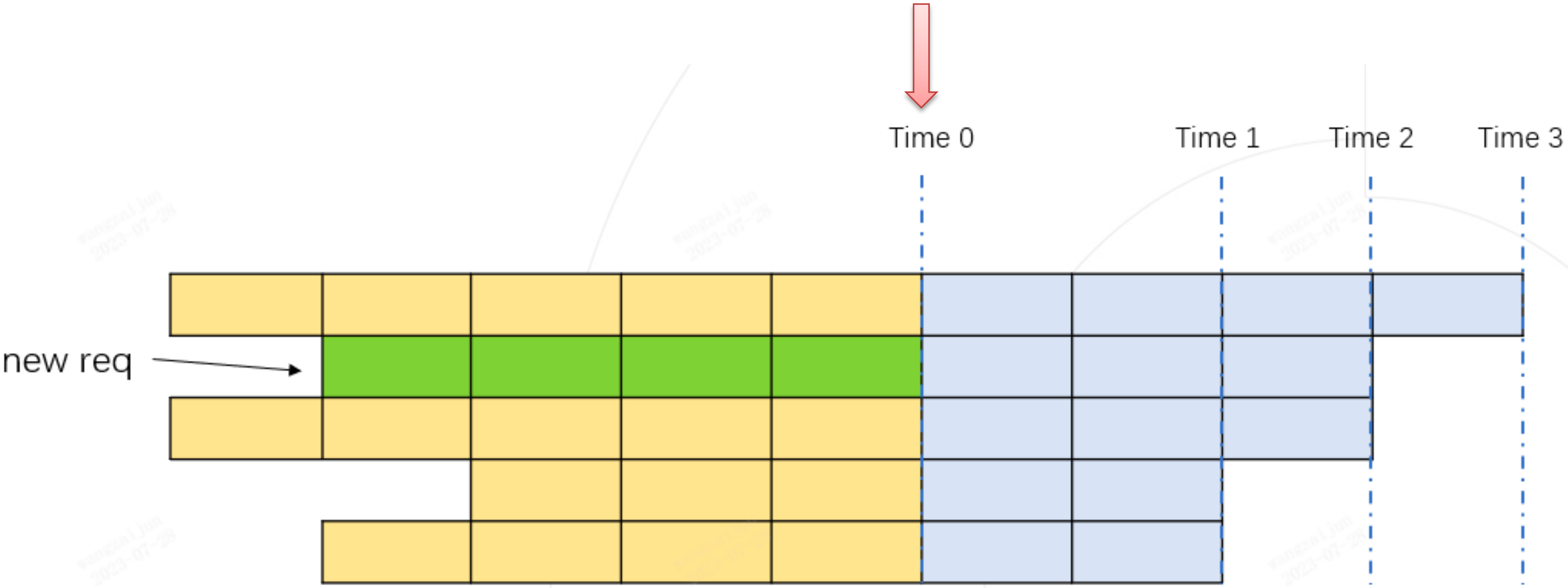


LightLLM – Token Attention

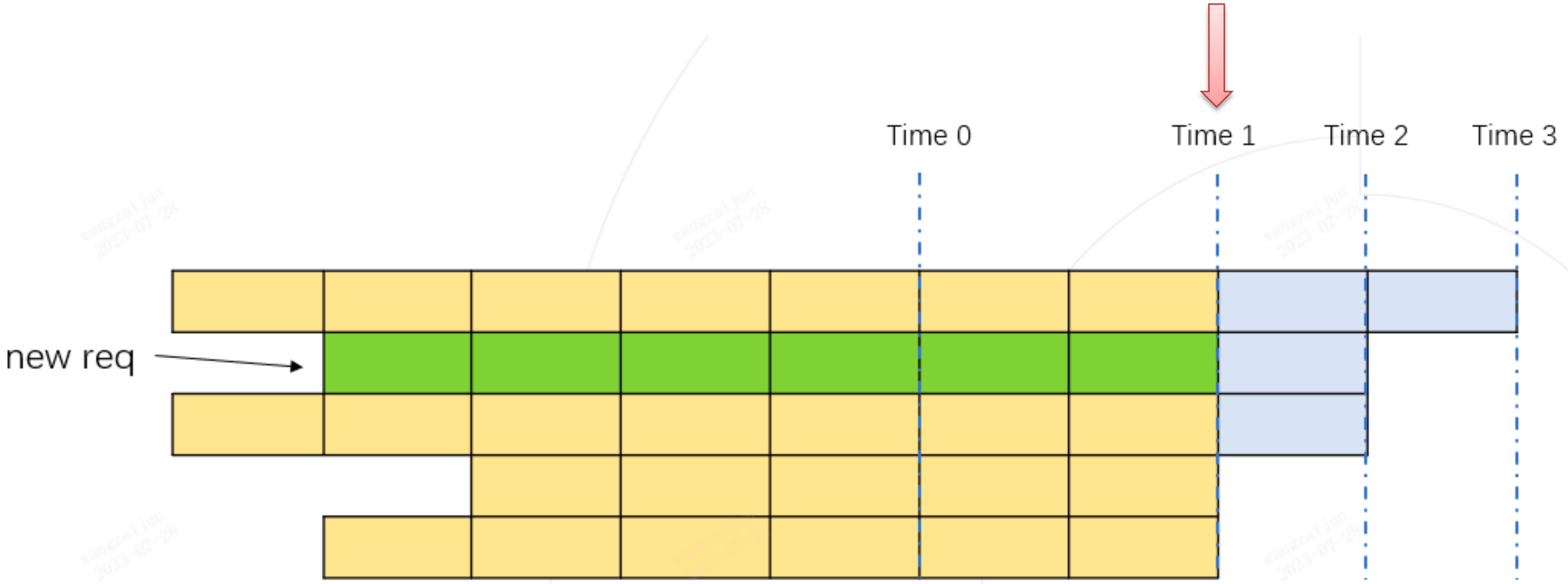
5. Merge a new seq3



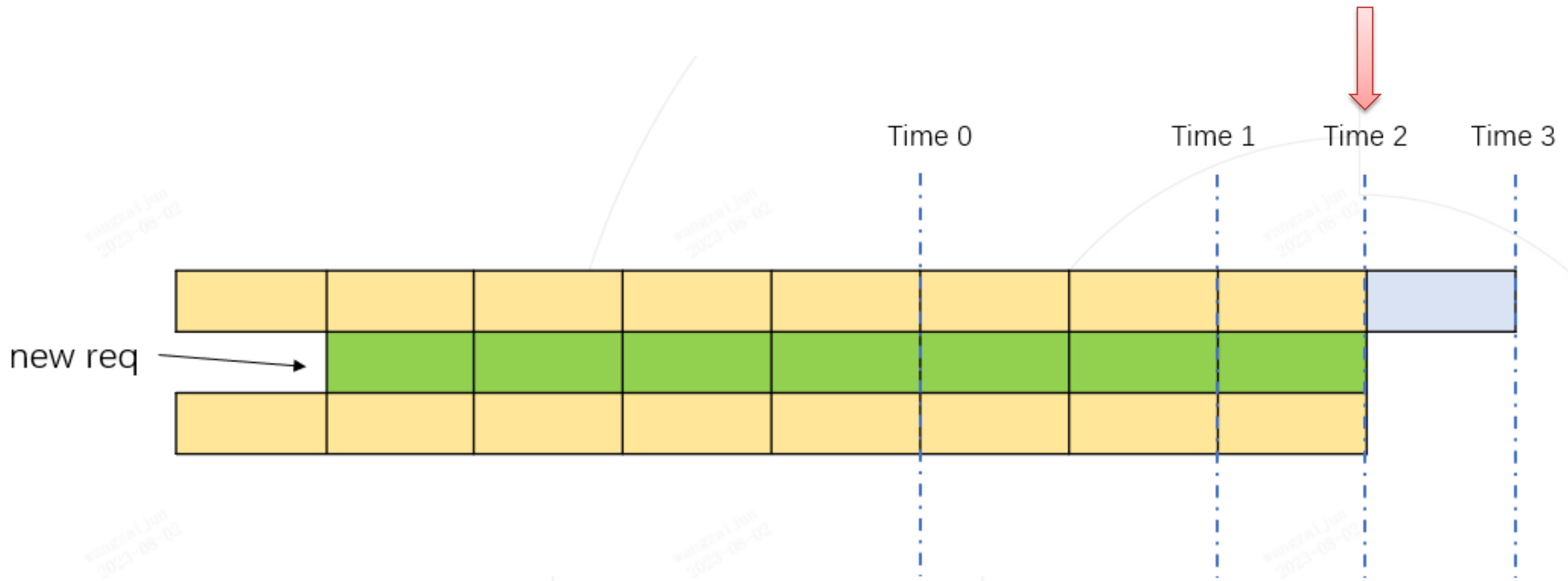
LightLLM – Efficient Router



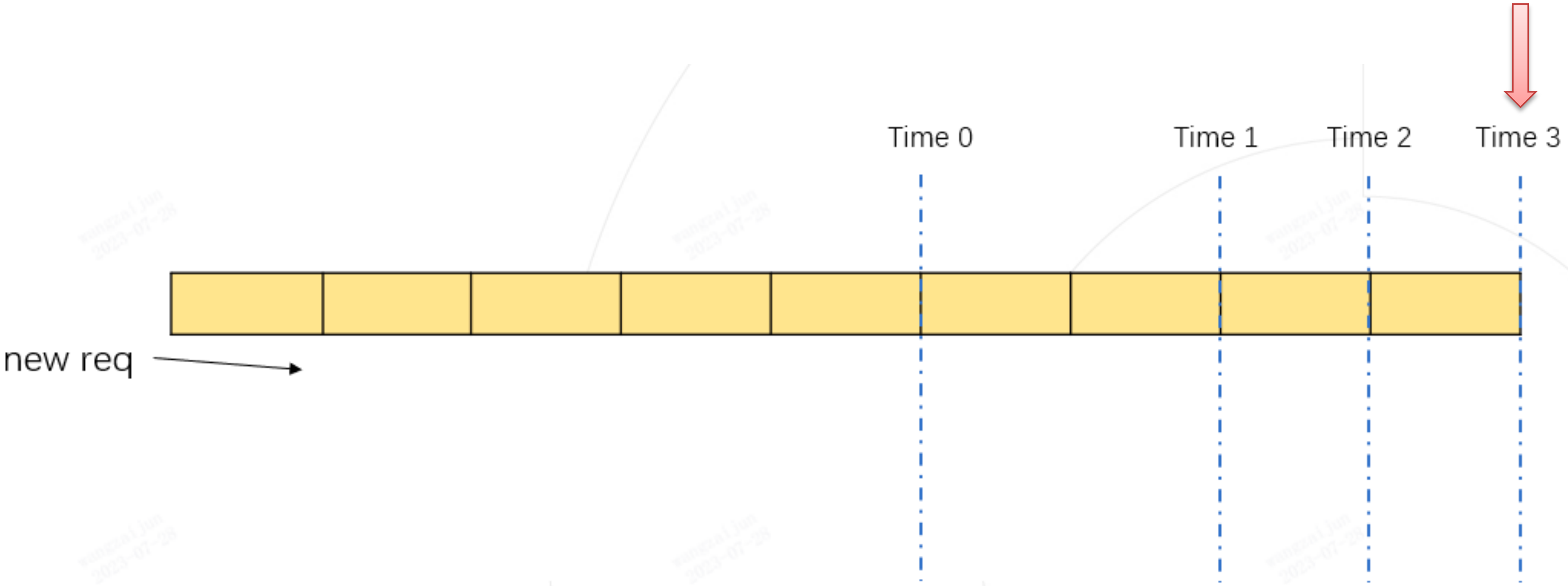
LightLLM – Efficient Router



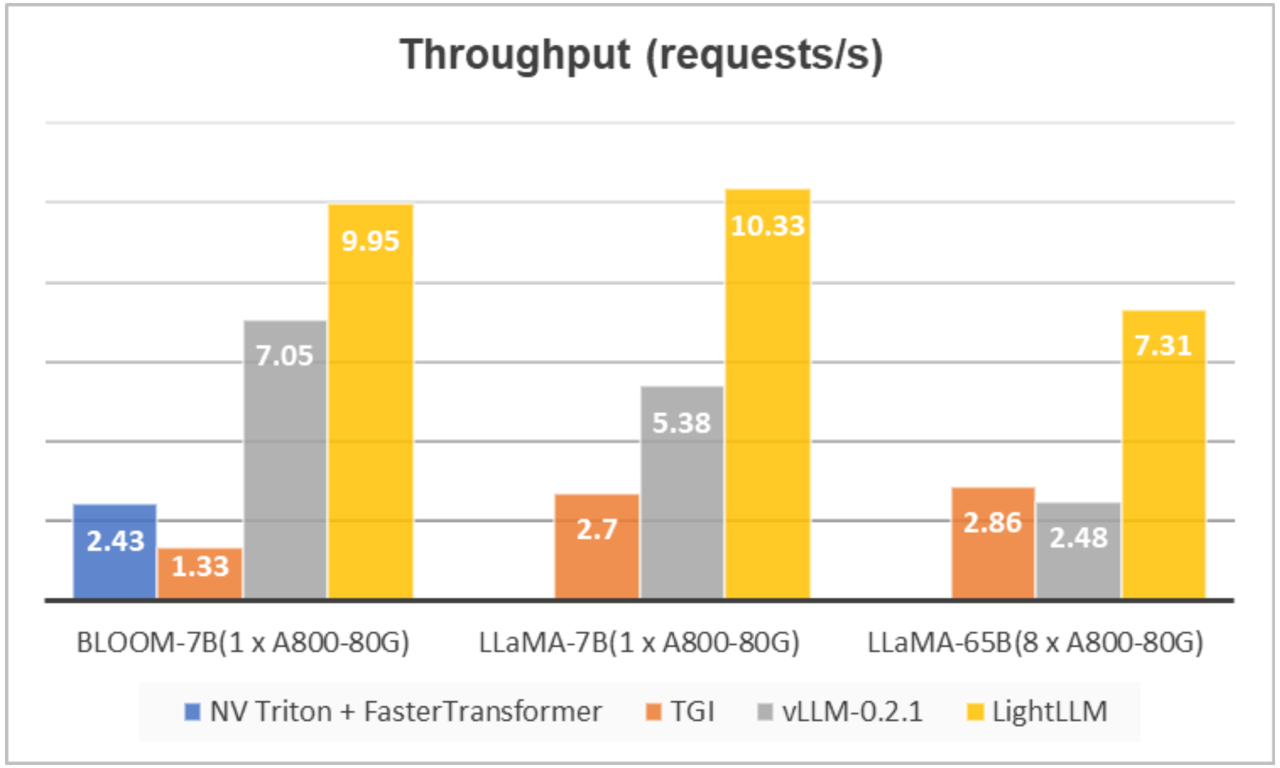
LightLLM – Efficient Router

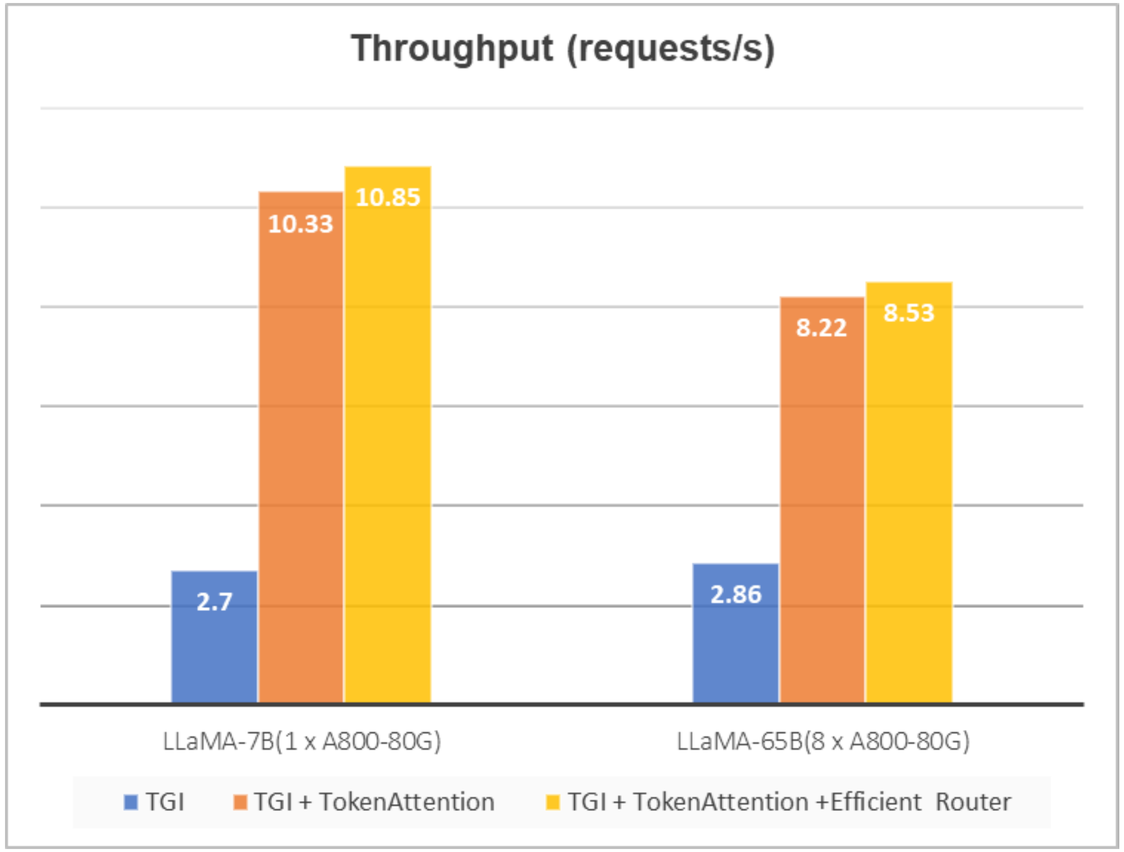


LightLLM – Efficient Router



LightLLM – Performance





LightLLM – Code

Run web server using docker:

```
pip install -r requirements.txt
docker pull ghcr.io/modeltc/lightllm:main
docker run -it --gpus all -p 8080:8080                  \
        --shm-size 1g -v your_local_path:/data/         \
        ghcr.io/modeltc/lightllm:main /bin/bash
python setup.py install
pip install triton==2.0.0.dev20221202
python -m lightllm.server.api_server --model_dir /path/llama-7B     \
                                     --host 0.0.0.0                 \
                                     --port 8080                    \
                                     --tp 1                         \
                                     --max_total_token_num 120000
```

Make queries:

```
import time
import requests
import json

url = 'http://localhost:8080/generate'
headers = {'Content-Type': 'application/json'}
data = {
    'inputs': 'Funniest joke ever:',
    "parameters": {
        'do_sample': False,
        'ignore_eos': False,
        'max_new_tokens': 1024,
    }
}
response = requests.post(url, headers=headers, data=json.dumps(data))
if response.status_code == 200:
    print(response.json())
else:
    print('Error:', response.status_code, response.text)
```

References

- https://a16z.com/emerging-architectures-for-llm-applications/
- https://superwise.ai/blog/considerations-best-practices-for-llm-architectures/
- https://huyenchip.com/2023/04/11/llm-engineering.html
- https://betterprogramming.pub/frameworks-for-serving-llms-60b7f7b23407
- https://developer.nvidia.com/blog/fast-and-scalable-ai-model-deployment-with-nvidia-triton-inference-server/
- https://huggingface.co/docs/text-generation-inference/en/index
- https://docs.databricks.com/en/workflows/index.html
- https://www.kdnuggets.com/2023/06/vector-databases-important-llms.html
- https://spotintelligence.com/2023/11/17/llm-orchestration-frameworks/
- Yu, G. I., Jeong, J. S., Kim, G. W., Kim, S., & Chun, B. G. (2022). Orca: A distributed serving system for {Transformer-Based} generative models. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22) (pp. 521-538).