# CS6216 Advanced Topics in Machine Learning (Systems)

# Serving LLMs

Yao LU

02 Oct 2024

National University of Singapore
School of Computing

# From LLMs to serving systems

Chef
(LLM)

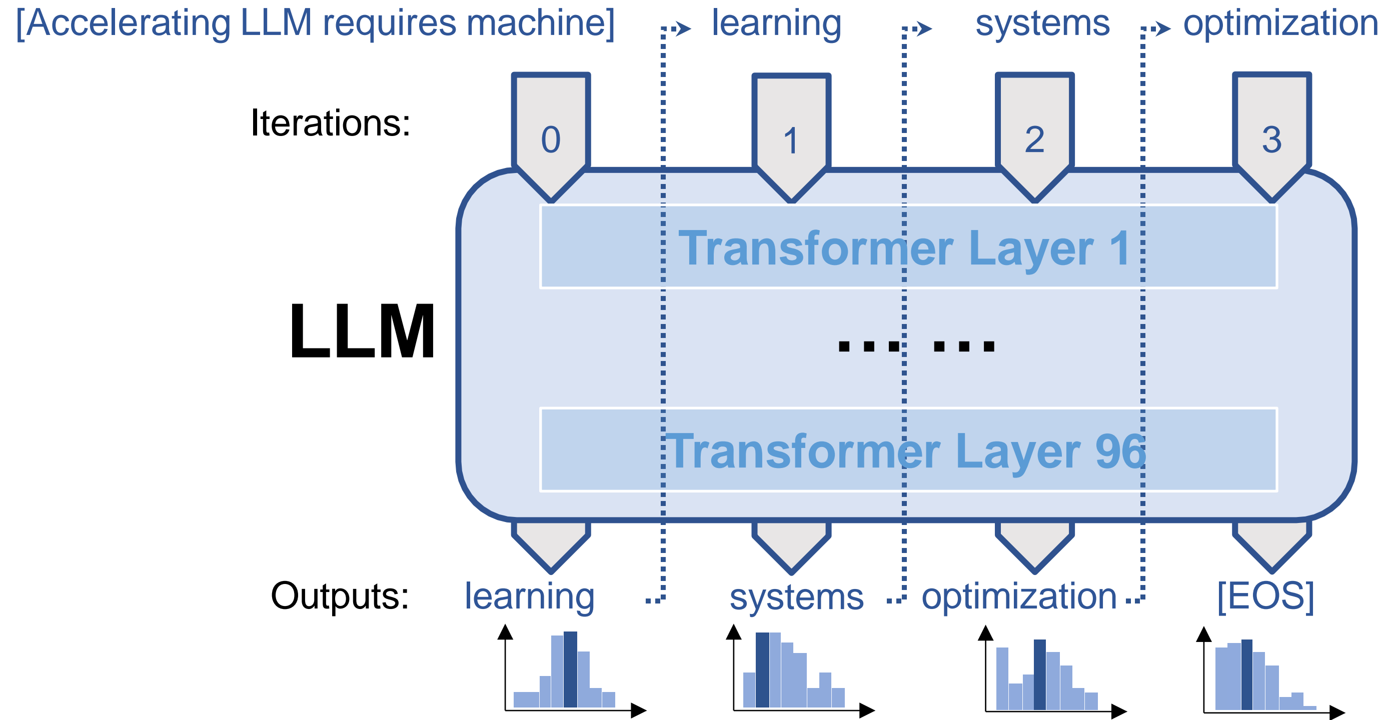
Restaurant
(serving systems)

- From LLM inference to a full-fledged system
  - Queueing, routing, batching,
  - Pricing & accounting,
  - Perf monitoring & optimization etc.

# Outline: LLMs serving techniques

- LLM decoding & system design
- Model quantization
- Continuous batching
- Speculative decoding
- Overall goals:
  - Improve latency, throughput, memory consumption, generalizability, ..

# Recall: LLM incremental decoding

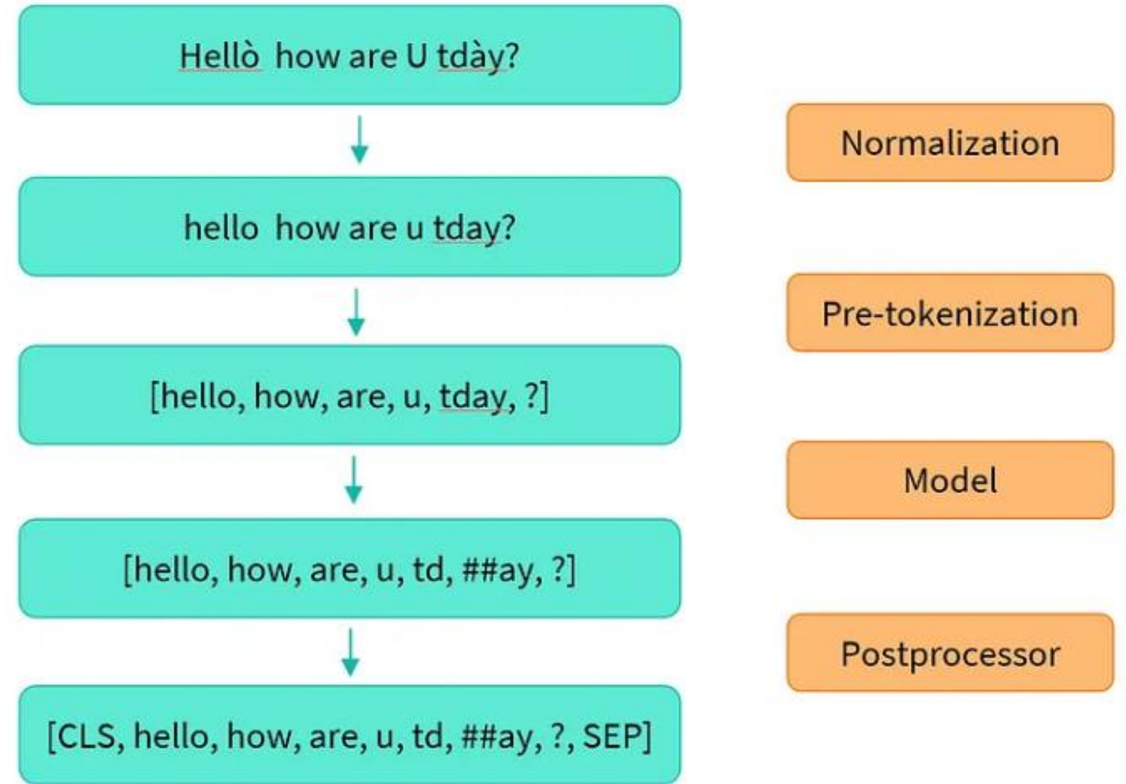



**Main issues**:

- Limited degree of parallelism → underutilized GPU resources
- Need all parameters to decode a token → bottlenecked by GPU memory access

# Tokenizer

- **Normalization:** cleaning up
- **Pre-tokenization:** splitting
- **Modeling:** mapping (sub)tokens
- **Postprocessor:** adding special tokens



# Modeling: Byte Pair Encoding (BPE)

- **Key idea:**
  - Common words are represented in the vocabulary as a single token
  - Rare words are broken down into two or more subword tokens
- **Example:**

| | |
|---|---|
| aaabdaaabac | Z=aa |
| → ZabdZabac | Y=ab |
| → ZYdZYac | X=ZY |
| → XdXac | |

- **Algorithm:**

Recursively find the most frequent (byte pair) and merge them

https://towardsdatascience.com/byte-pair-encoding-subword-based-tokenization-algorithm-77828a70bee0

# Modeling: Byte Pair Encoding (BPE)

## Input corpus:

{“old</w>”: 7, “older</w>”: 3, “finest</w>”: 9, “lowest</w>”: 4}

</w> is word boundary

## Steps:

- List and find the most frequent (byte) pair
- Merge and create a new token
- Update the frequency counts in dictionary

## Stopping criteria:

- Word level: reaching </w>
- Overall: reaching token count

| Number | Token | Frequency |
|---|---|---|
| 1 | </w> | 23 |
| 2 | o | 14 |
| 3 | l | 14 |
| 4 | d | 10 |
| 5 | e | 16 |
| 6 | r | 3 |
| 7 | f | 9 |
| 8 | i | 9 |
| 9 | n | 9 |
| 10 | s | 13 |
| 11 | t | 13 |
| 12 | w | 4 |

# Modeling: Byte Pair Encoding (BPE)

**Most frequent: es**

## Input corpus:

{“old</w>”: 7, “older</w>”: 3, “finest</w>”: 9, “lowest</w>”: 4}

</w> is word boundary

## Steps:

- List and find the most frequent (byte) pair
- Merge and create a new token
- Update the frequency counts in dictionary

## Stopping criteria:

- Word level: reaching </w>
- Overall: reaching token count

| Number | Token | Frequency |
|---|---|---|
| 1 | </w> | 23 |
| 2 | o | 14 |
| 3 | l | 14 |
| 4 | d | 10 |
| 5 | e | 16 - 13 = 3 |
| 6 | r | 3 |
| 7 | f | 9 |
| 8 | i | 9 |
| 9 | n | 9 |
| 10 | s | 13 - 13 = 0 |
| 11 | t | 13 |
| 12 | w | 4 |
| 13 | es | 9 + 4 = 13 |

# Modeling: Byte Pair Encoding (BPE)

**Most frequent: est**

## Input corpus:

{“old</w>”: 7, “older</w>”: 3, “finest</w>”: 9, “lowest</w>”: 4}

</w> is word boundary

## Steps:

- List and find the most frequent (byte) pair
- Merge and create a new token
- Update the frequency counts in dictionary

## Stopping criteria:

- Word level: reaching </w>
- Overall: reaching token count

| Number | Token | Frequency |
|---|---|---|
| 1 | </w> | 23 |
| 2 | o | 14 |
| 3 | l | 14 |
| 4 | d | 10 |
| 5 | e | 16 - 13 = 3 |
| 6 | r | 3 |
| 7 | f | 9 |
| 8 | i | 9 |
| 9 | n | 9 |
| 10 | s | 13 - 13 = 0 |
| 11 | t | 13 - 13 = 0 |
| 12 | w | 4 |
| 13 | es | 9 + 4 = 13 - 13 = 0 |
| 14 | est | 13 |

# Modeling: Byte Pair Encoding (BPE)

**Most frequent: est</w>**

## Input corpus:

{“old</w>”: 7, “older</w>”: 3, “finest</w>”: 9, “lowest</w>”: 4}

</w> is word boundary

## Steps:

- List and find the most frequent (byte) pair
- Merge and create a new token
- Update the frequency counts in dictionary

## Stopping criteria:

- Word level: reaching </w>
- Overall: reaching token count

| Number | Token | Frequency |
| --- | --- | --- |
| 1 | </w> | 23 - 13 = 10 |
| 2 | o | 14 |
| 3 | l | 14 |
| 4 | d | 10 |
| 5 | e | 16 - 13 = 3 |
| 6 | r | 3 |
| 7 | f | 9 |
| 8 | i | 9 |
| 9 | n | 9 |
| 10 | s | 13 - 13 = 0 |
| 11 | t | 13 - 13 = 0 |
| 12 | w | 4 |
| 13 | es | 9 + 4 = 13 - 13 = 0 |
| 14 | est | 13 - 13 = 0 |
| 15 | est</w> | 13 |

# Modeling: Byte Pair Encoding (BPE)

**Most frequent: ol**

## Input corpus:

{“old</w>”: 7, “older</w>”: 3, “finest</w>”: 9, “lowest</w>”: 4}

</w> is word boundary

## Steps:

- List and find the most frequent (byte) pair
- Merge and create a new token
- Update the frequency counts in dictionary

## Stopping criteria:

- Word level: reaching </w>
- Overall: reaching token count

| Number | Token | Frequency |
|---|---|---|
| 1 | </w> | 23 |
| 2 | o | 14 - 10 = 4 |
| 3 | l | 14 - 10 = 4 |
| 4 | d | 10 |
| 5 | e | 16 - 13 = 3 |
| 6 | r | 3 |
| 7 | f | 9 |
| 8 | i | 9 |
| 9 | n | 9 |
| 10 | s | 13 - 13 = 0 |
| 11 | t | 13 - 13 = 0 |
| 12 | w | 4 |
| 13 | es | 9 + 4 = 13 - 13 = 0 |
| 14 | est | 13 |
| 15 | ol | 7 + 3 = 10 |

# Modeling: Byte Pair Encoding (BPE)

**Most frequent: old**

## Input corpus:

{“old</w>”: 7, “older</w>”: 3, “finest</w>”: 9, “lowest</w>”: 4}

</w> is word boundary

## Steps:

- List and find the most frequent (byte) pair
- Merge and create a new token
- Update the frequency counts in dictionary

## Stopping criteria:

- Word level: reaching </w>
- Overall: reaching token count

| Number | Token | Frequency |
|---|---|---|
| 1 | </w> | 23 - 13 = 10 |
| 2 | o | 14 - 10 = 4 |
| 3 | l | 14 - 10 = 4 |
| 4 | d | 10 - 10 = 0 |
| 5 | e | 16 - 13 = 3 |
| 6 | r | 3 |
| 7 | f | 9 |
| 8 | i | 9 |
| 9 | n | 9 |
| 10 | s | 13 - 13 = 0 |
| 11 | t | 13 - 13 = 0 |
| 12 | w | 4 |
| 13 | es | 9 + 4 = 13 - 13 = 0 |
| 14 | est | 13 - 13 = 0 |
| 15 | est</w> | 13 |
| 16 | ol | 7 + 3 = 10 - 10 = 0 |
| 17 | old | 7 + 3 = 10 |

# Modeling: Byte Pair Encoding (BPE)

## Input corpus:

{“old</w>”: 7, “older</w>”: 3, “finest</w>”: 9, “lowest</w>”: 4}

</w> is word boundary

## Encoding and decoding:

**Decoding**: straightforward

[“the</w>”, “high”, “est</w>”, “range</w>”, “in</w>”, “Seattle</w>”]

→ the highest range in Seattle.

**Encoding**:

- Iteratively replace tokens from longest to shortest
- Replace leftovers with OOV token

## Cleanup dictionary

| Number | Token | Frequency |
|---|---|---|
| 1 | </w> | 10 |
| 2 | o | 4 |
| 3 | l | 4 |
| 4 | e | 3 |
| 5 | r | 3 |
| 6 | f | 9 |
| 7 | i | 9 |
| 8 | n | 9 |
| 9 | w | 4 |
| 10 | est</w> | 13 |
| 11 | old | 10 |

# LLM decoding

```
mat: 0.6
couch: 0.2
bed: 0.1
chair: 0.05
car: 0.003
bike: 0.01
bucket: 0.3
......
```

- LLM decoding is like a pottery wheel
- **Greedy decoding:** always pick the highest prob
- **Sampling-based decoding:** use top-k, p, temperature to “shape” the pottery
- **Beam search:** maximize overall prob in a search window

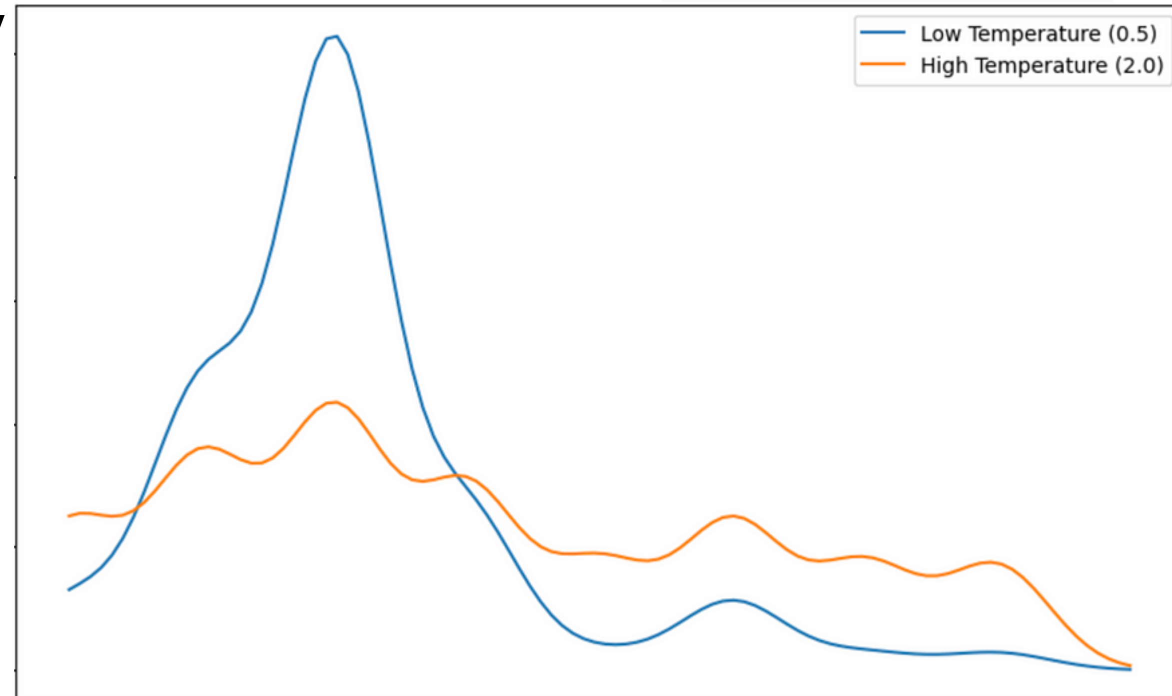
# LLM decoding: sampling-based methods

```
mat: 0.6
couch: 0.2
bed: 0.1
chair: 0.05
car: 0.003
bike: 0.01
bucket: 0.3
......
```

- Top-K limits each generation within the top K choices
- Top-P filters choices (keep those at least probability P)

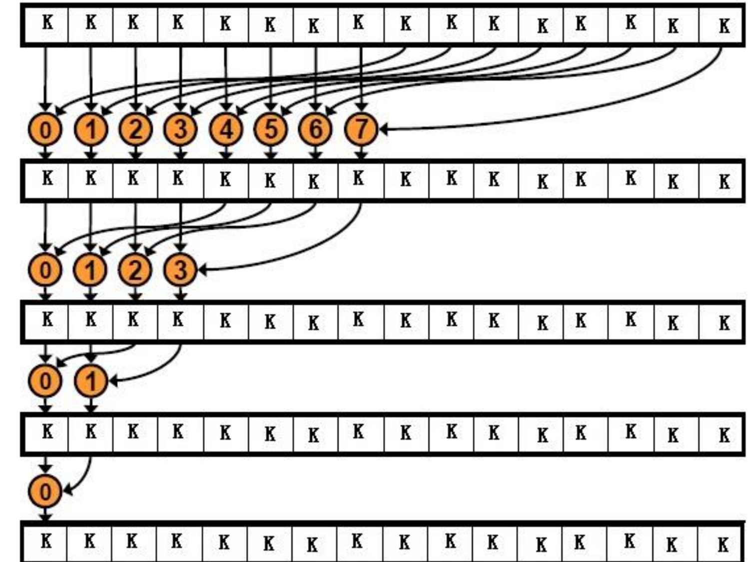
- Temperature adjusts the probability scores: **log_prob_scaled = log_prob / temperature**

- Application order:
  Temperature → top-K → top-P



# LLM decoding: sampling-based methods

- Top-K complexity: O(k log n)
  - n could be tens of thousands or more
  - Similar for softmax
- Techniques to accelerate top-k or softmax
  - Staged, parallel top-k on GPUs
  - Advanced sampling algorithms
    - Gumbel-max sampling
    - Hierarchical softmax
    - Importance sampling



# Constrained decoding

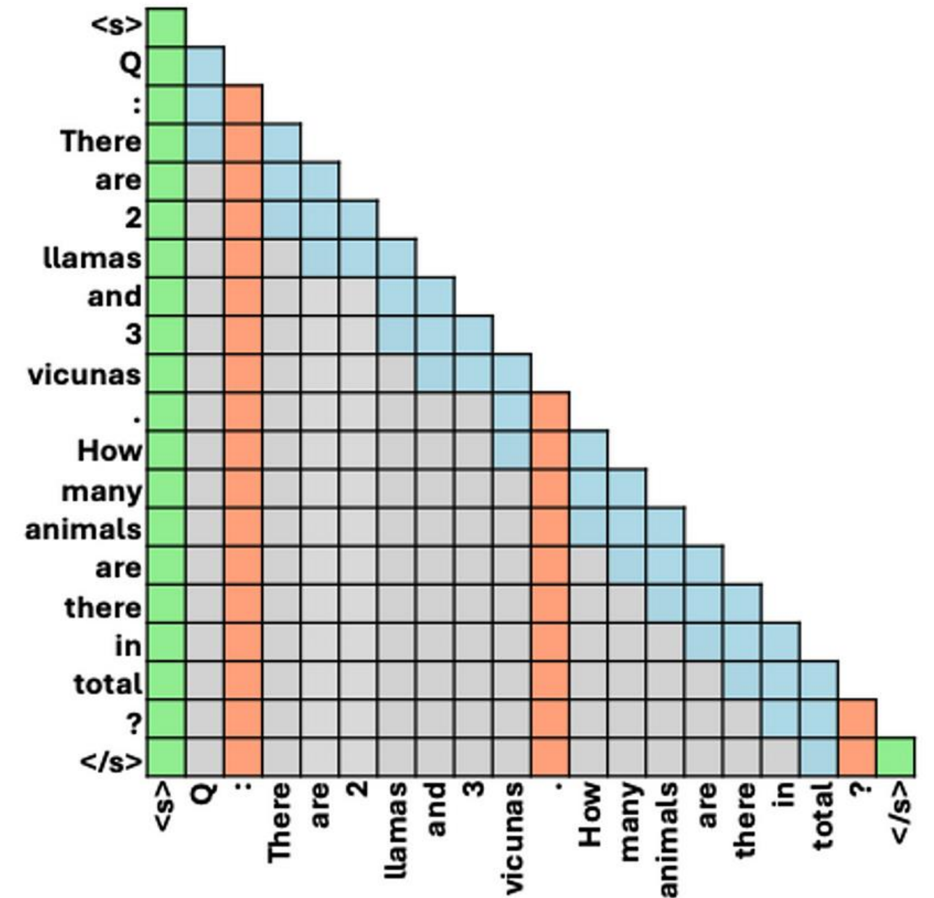
- Sampling-based decoding does not consider semantics

- Constrained decoding can use
  - Grammar
  - Regex
  - Choices
  - Data type, length

- Different implementations: Finite State Machine (FSM), masking, etc.

# KV cache management

- KV cache requirement in ~1MB/token
  - $2 \cdot b \cdot t \cdot n_{layers} \cdot n_{heads} \cdot d_{head} \cdot p_a$
- Strategies include
  - Novel attention architectures
  - Efficient memory management
  - Cache compression
  - Evict to CPU/disk



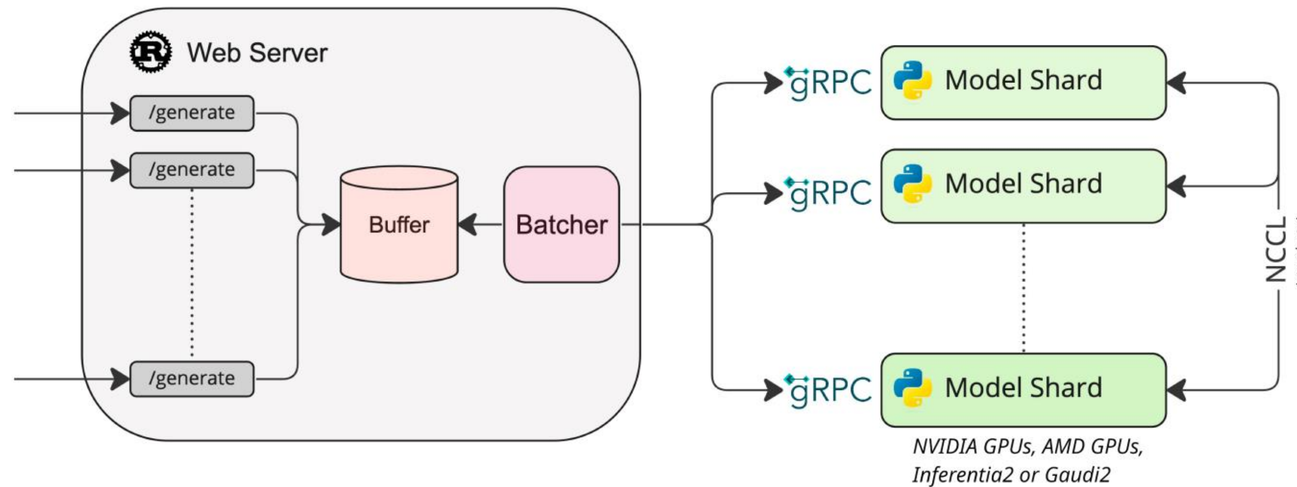

(FastGen) Example of set of compression policies: Special tokens (green) + Punctuation tokens (orange) + Local attention (blue). Discarded tokens are colored in gray.

# Stopping criteria in LLM generation

- **Stopping word:** a special token <EOS>, <s> etc.
- **Stopping string:** a sequence of tokens
- **Max token count:** # of tokens generated so far

# Serving system architecture

Text Generation Inference
Fast optimized inference for LLMs




**Key ideas:**

- Each model = 1 container pod
- User ⟷ server ⟷ model shard (Restful API, 1:1 between user and server; gRPC, 1:n between server and model shard)
- Pytorch & Huggingface ecosystem
- Model shard in Python, server in Rust

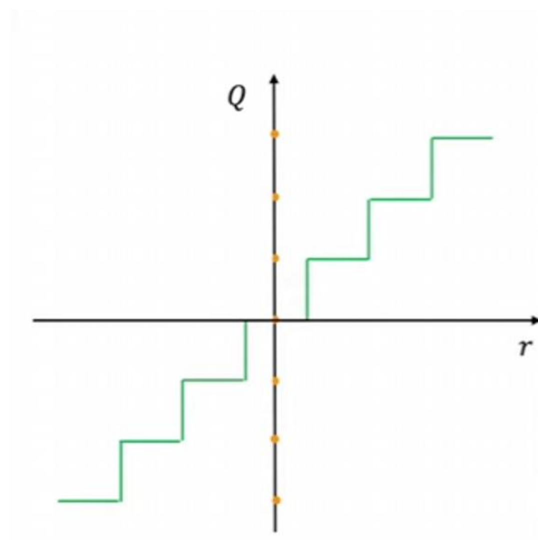
# Outline: LLMs serving techniques

- LLM decoding & system design
- Model quantization
- Continuous batching
- Speculative decoding

# Model quantization

- DNNs originally use FP32 precision
  - Continuous values → FP32 quantization
  - In comparison, images use 3 x [0,256] pixels

- Convert models to lower precisions
  - ➢ Reduce memory usages & deploy on low-resource devices
  - ➢ Improve training & inference speed
  - ➢ Extreme cases: use bitwise operators
  - ➢ **(But) At the tradeoff of accuracy lost**




**Rounding:** find the nearest integer
1.8 → 2,   1.2 → 1

**Truncating:** remove the decimal part
1.8 → 1,   1.2 →  1

# Floating point representations

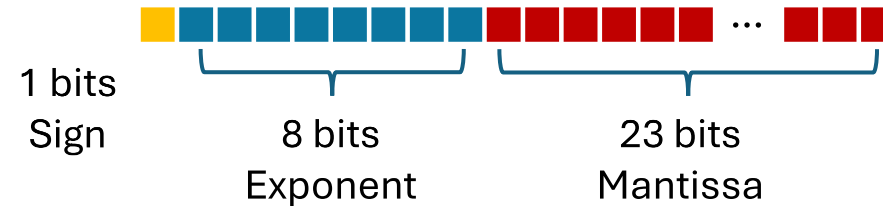
- Examples:

**Original value 0.0001**
FP16: 0.00010001659393 (Binary: 0|00001|1010001110, Hex: 068E)
BF16: 0.00010013580322 (Binary: 0|01110001|1010010, Hex: 38D2)

**Original value 1e-08**
FP16: 0.00000000000000 (Binary: 0|00000|0000000000, Hex: 0000)
BF16: 0.00000001001172 (Binary: 0|01100100|0101100, Hex: 322C)

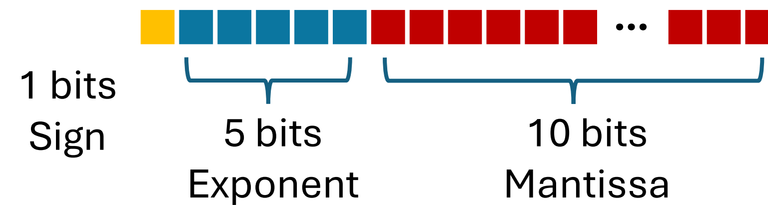
**Original value 100000.00001**
FP16: inf (Binary: 0|11111|0000000000, Hex: 7C00)
BF16: 99840.0000000000000 (Binary: 0|10001111|1000011, Hex: 47C3)

1 bits
Sign



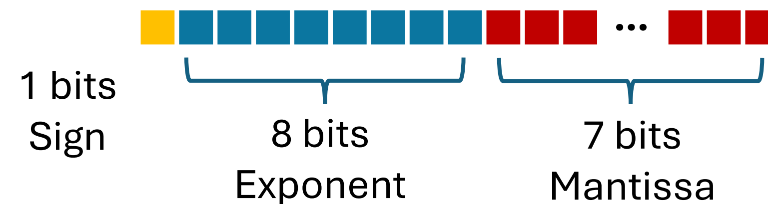

FP32

1 bits
Sign




FP16

1 bits
Sign



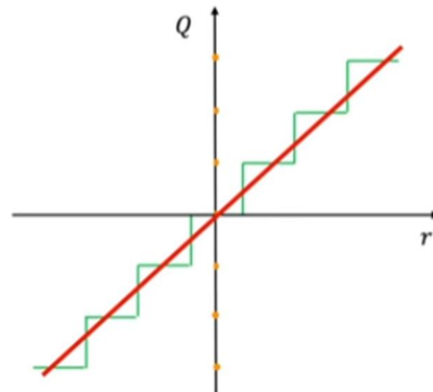

BF16,

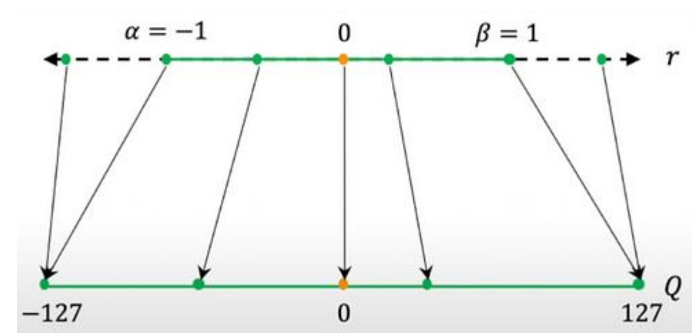
B=Google brain

BF16 provides a wider range at a cost of some precision → balance between range & numerical stability

# Quantization

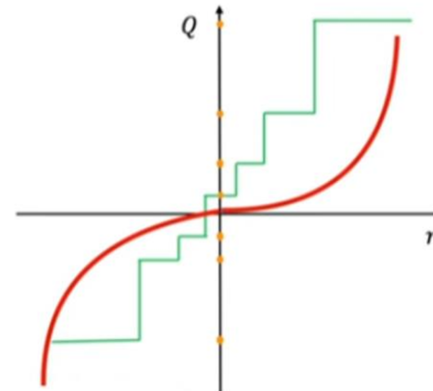
- Uniform, symmetric inputs

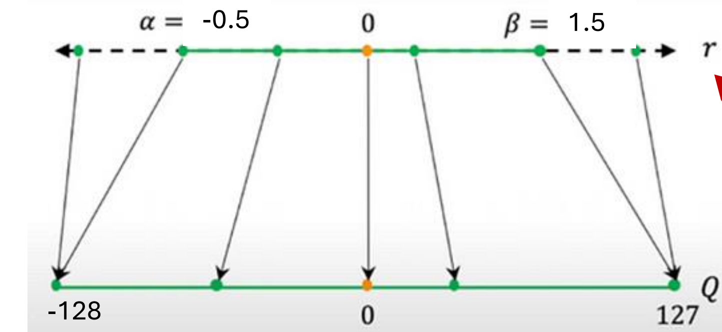



$$Q = round\left(\frac{r}{s}\right)$$

- Non-uniform, asymmetric inputs

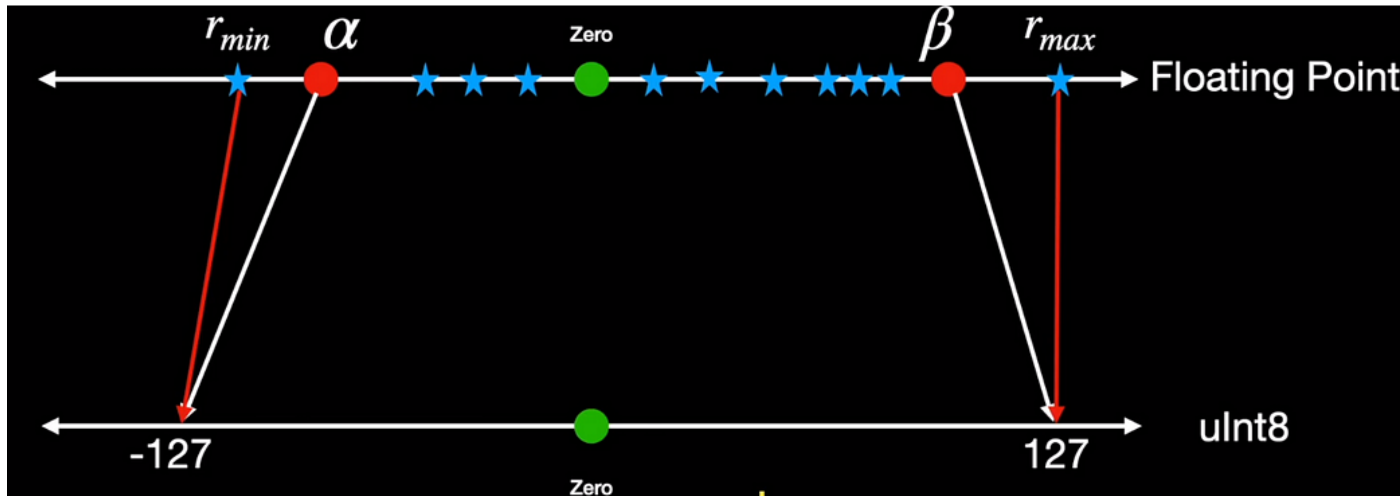



$$Q = round\left(\frac{r}{s}\right)+Z$$

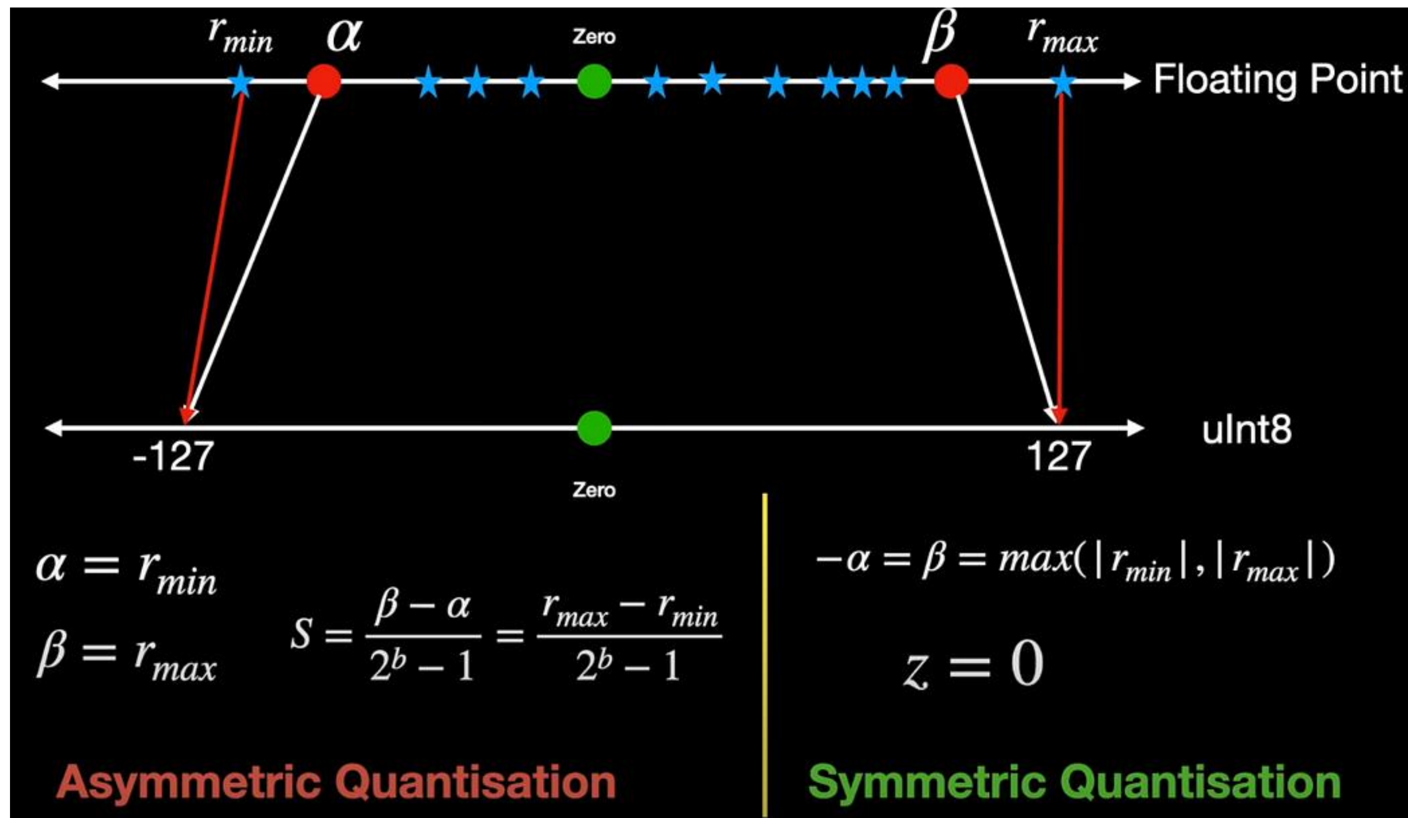
Dequantization $\tilde{r} = S(Q + Z)$,
Perplexity, error $\epsilon = \tilde{r} - \mathrm{r}$

# Calibration: choosing scale and zero factor

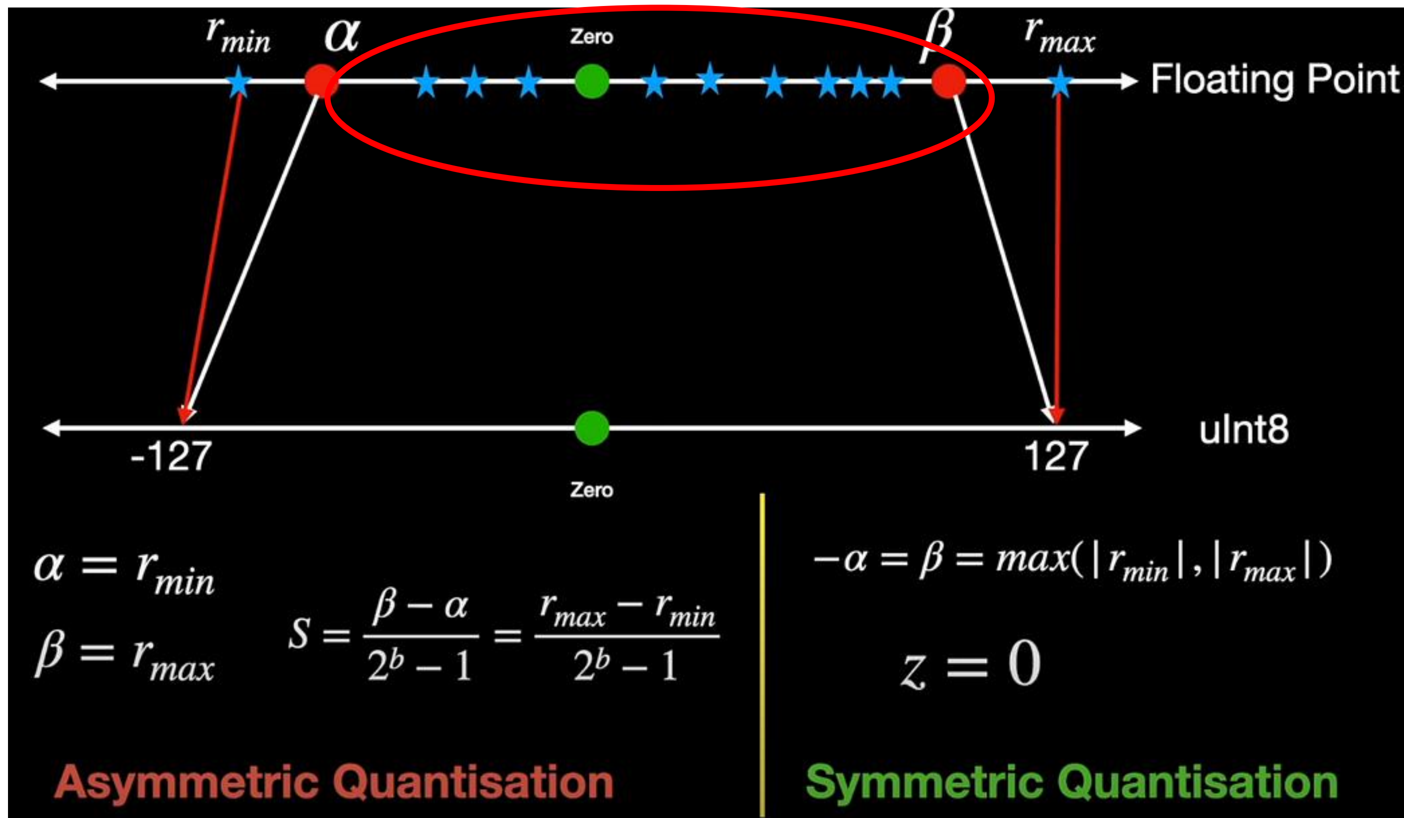



https://medium.com/@AIBites/model-quantization-in-deep-neural-networks-81df49f3c7d8

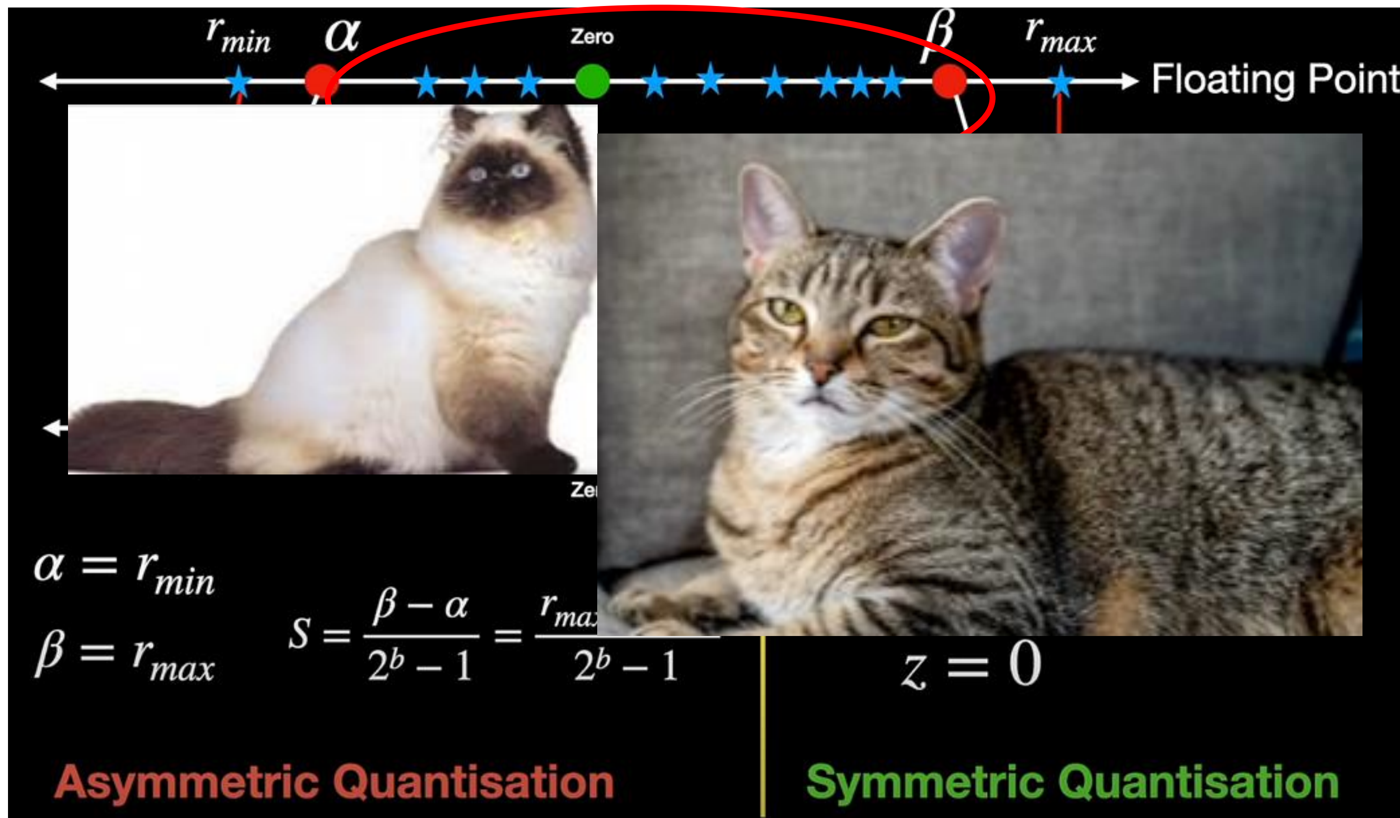
# Calibration: choosing scale and zero factor



# Calibration: rectifying skews



# Calibration: rectifying skews




**When & How to calibrate?**

During or after training?

Data skew is unknown a priori

# Quantization modes

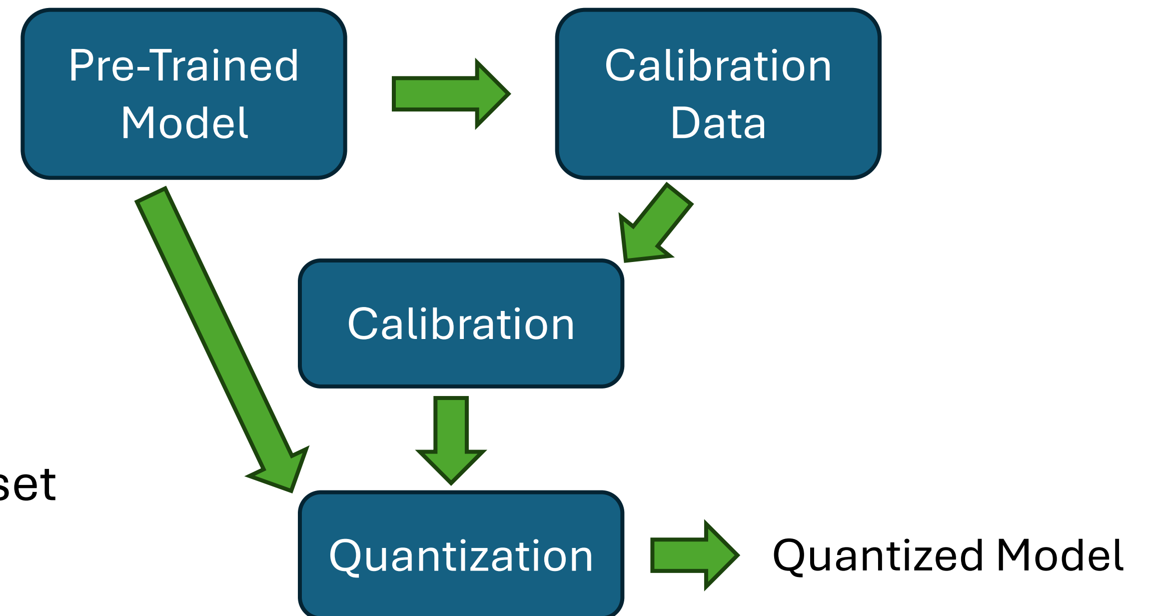
- Post Training Quantization (PTQ)
  - **Weight-only quantization:**

    Inflate model weights during computation

    May not need calibration dataset
  - **Full quantization:**

    Weights + activation, need calibration dataset

    Calibrations include:
    - Output bias caused by quantization, add up to the final output
    - Weights, based on mean and variance before/after quantization

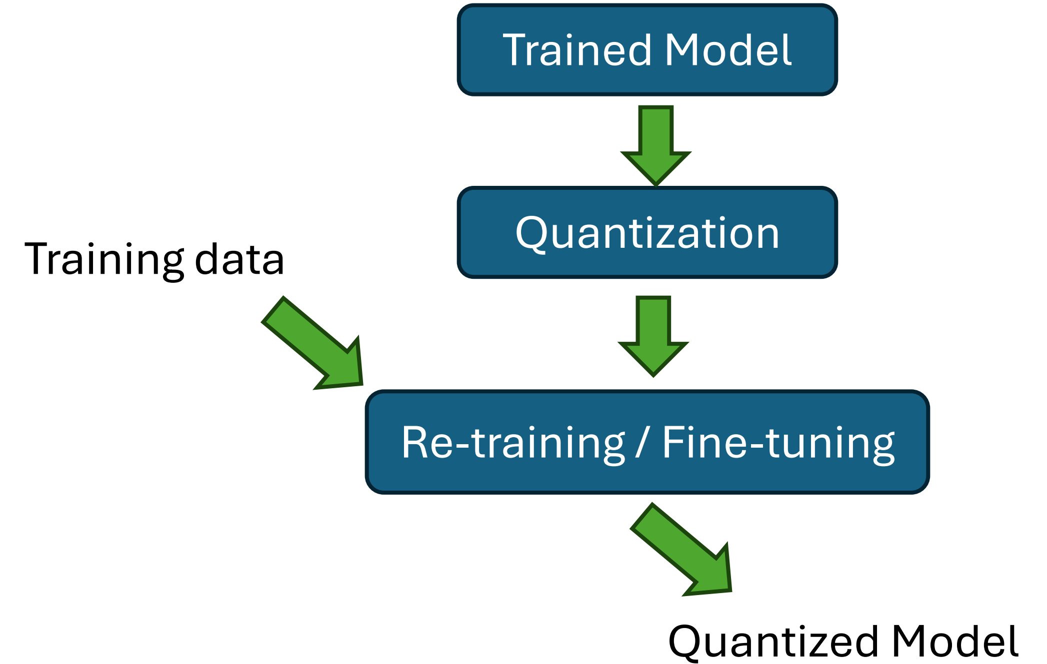


# Quantization modes

- Post Training Quantization (PTQ)
- Quantization Aware Training (QAT)
- Quantization Aware Fine-Tuning (QAF)
  - **Challenge:** quantization is not differentiable.
  - **Solution:**

    Insert fake quantization operators in the graph to compute statistics of the inputs

    Once the model is trained, update weights and remove the fake operators



# Quantization targets

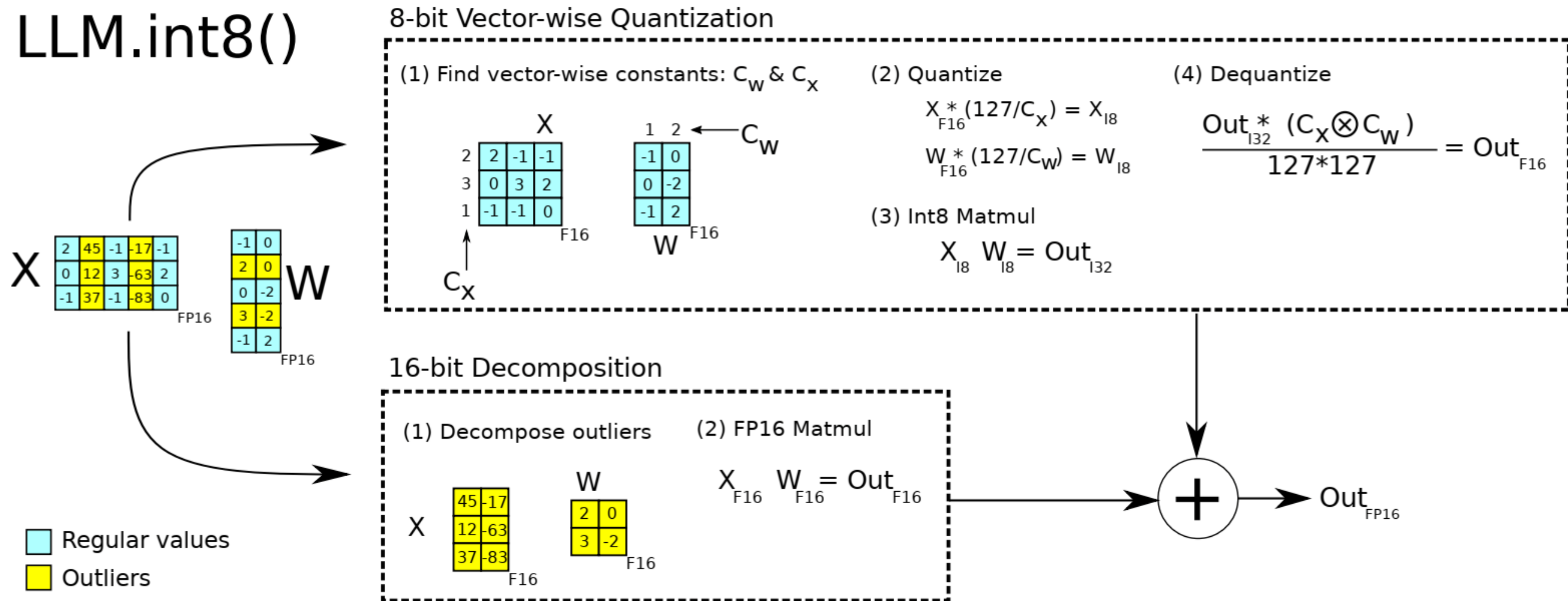
a) Weights (W): reduce model sizes and memory footprint

b) Activation (X): reduce memory footprint and improve speed

c) KV cache: improve throughput

d) Gradients: training only – reduce networking costs

# Weight-only quantization: LLM.int8()

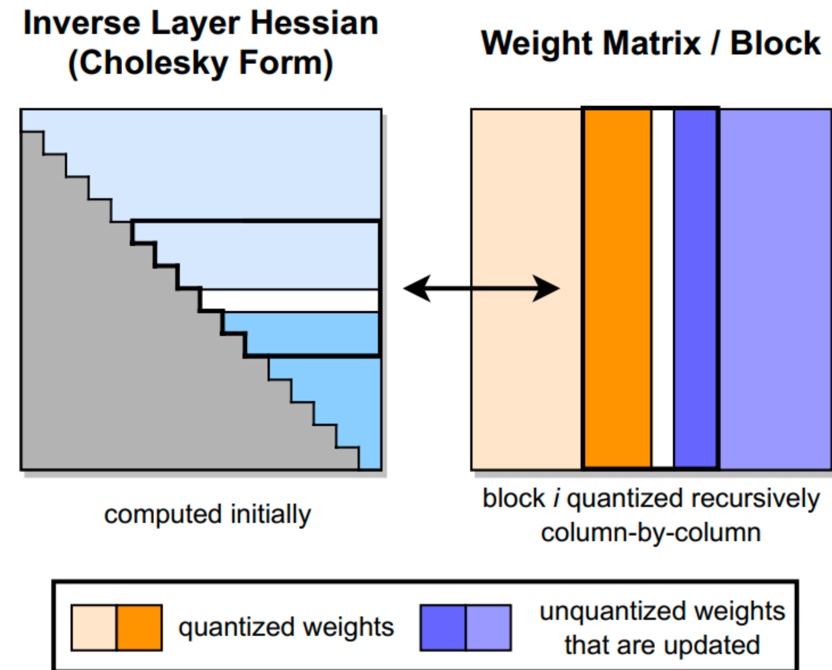



- Decompose the matrix
- Use 8bit quantization for the majority, 16bit for outliers

# Weight-only quantization: GPTQ

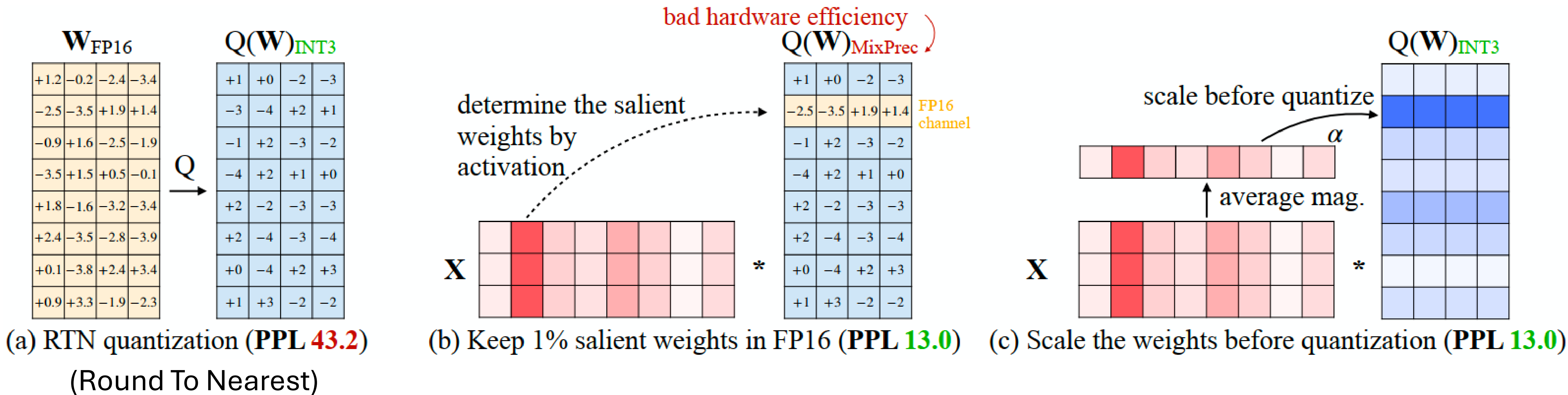
- Need calibration data

- Recursive process to
  - Quantize a block
  - Update the remaining weights to recover accuracy lost




GPTQ: Accurate Post-Training Quantization for Generative Pre-Trained Transformers. 2022

# Weight-only quantization: AWQ



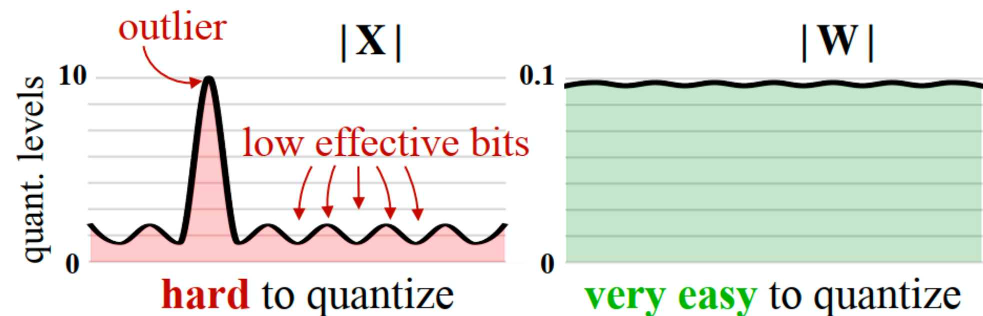

(Round To Nearest)

- Keep salient weights (by observing the activation distribution) in FP16 can greatly reduce quantization error
- Use per-channel scaling

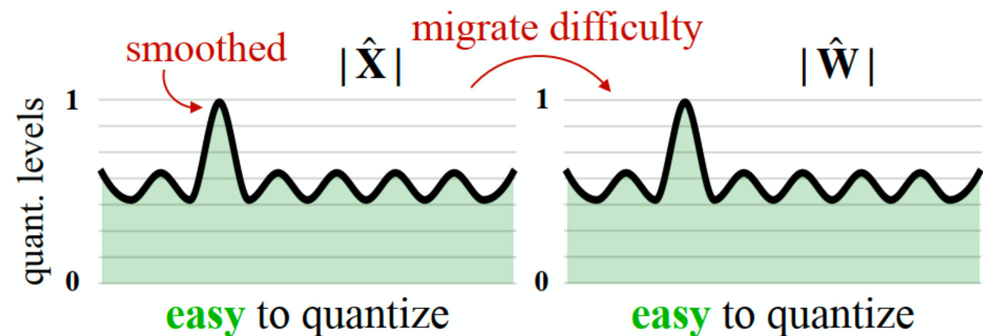
AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. 2023

# Full quantization: SmoothQuant

- Activations are harder to quantize
- Propose to smooth activations by transformation on the weights
- Use per-token and per-channel quantization



(a) Original



(b) SmoothQuant

@稀土掘金技术社区

Accurate and Efficient Post-Training Quantization for Large Language Models. 2022

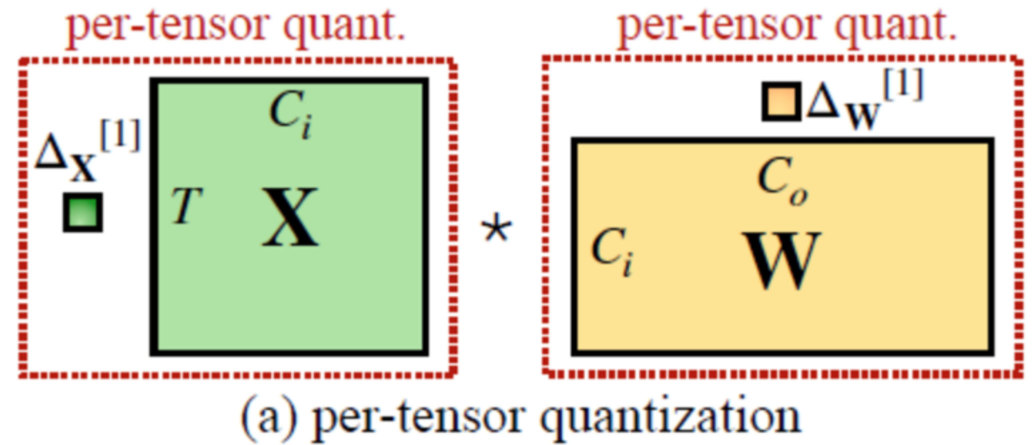
# Quantization granularity

a) **Per-tensor:** whole layer of input matrices

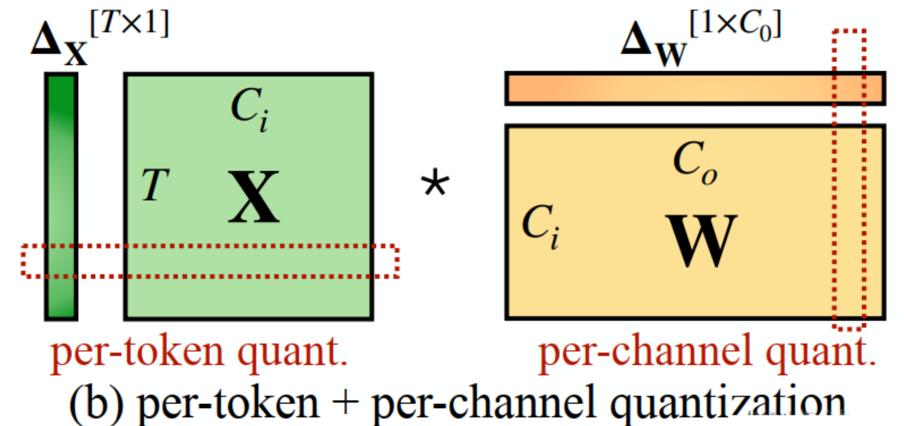
b) **Per-token & per-channel:** slices of input matrices

c) **Per-group:** combination of above

(b), (c) result in mixed-precision quantization schemes.



(a) per-tensor quantization

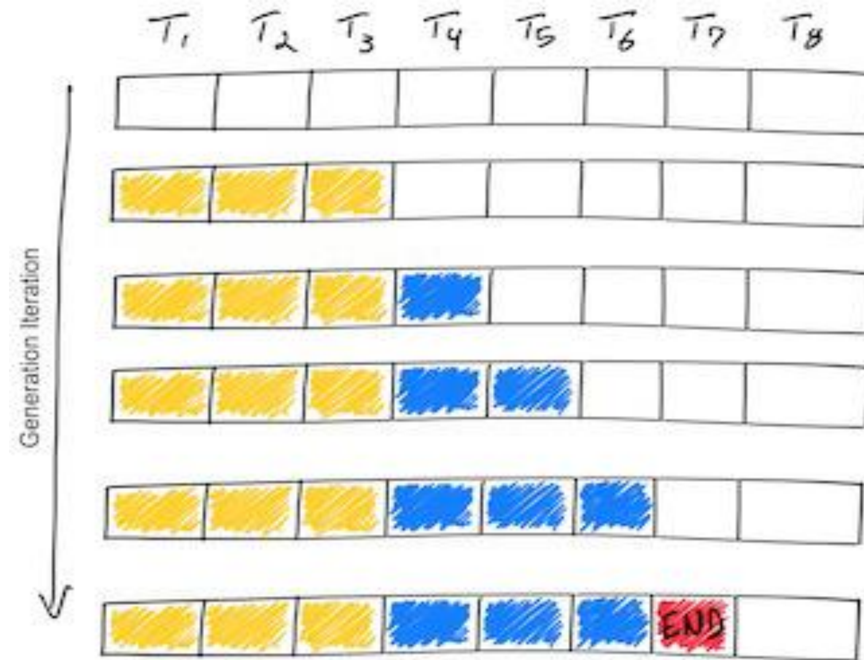


(b) per-token + per-channel quantization

# Outline: LLMs serving techniques

- LLM decoding & system design
- Model quantization
- Continuous batching
- Speculative decoding

# LLM decoding timeline



# Batching requests to improve GPU utilization

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ |
|---|---|---|---|---|---|---|---|
| $S_1$ | $S_1$ | $S_1$ | $S_1$ | | | | |
| $S_2$ | $S_2$ | $S_2$ | | | | | |
| $S_3$ | $S_3$ | $S_3$ | $S_3$ | | | | |
| $S_4$ | $S_4$ | $S_4$ | $S_4$ | $S_4$ | | | |

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ |
|---|---|---|---|---|---|---|---|
| $S_1$ | $S_1$ | $S_1$ | $S_1$ | $S_1$ | END | | |
| $S_2$ | $S_2$ | $S_2$ | $S_2$ | $S_2$ | $S_2$ | $S_2$ | END |
| $S_3$ | $S_3$ | $S_3$ | $S_3$ | END | | | |
| $S_4$ | $S_4$ | $S_4$ | $S_4$ | $S_4$ | $S_4$ | END | |

Issues with static batching:

- Requests may complete at different iterations
- Idle GPU cycles
- New requests cannot start immediately

# Continuous batching

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ |
|---|---|---|---|---|---|---|---|
| $S_1$ | $S_1$ | $S_1$ | $S_1$ | | | | |
| $S_2$ | $S_2$ | $S_2$ | | | | | |
| $S_3$ | $S_3$ | $S_3$ | $S_3$ | | | | |
| $S_4$ | $S_4$ | $S_4$ | $S_4$ | $S_4$ | | | |

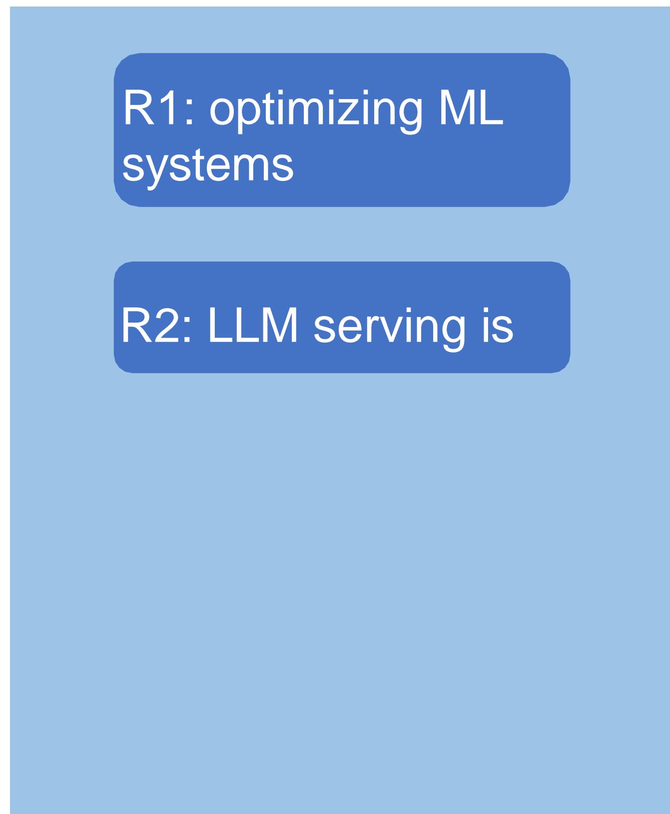
| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ |
|---|---|---|---|---|---|---|---|
| $S_1$ | $S_1$ | $S_1$ | $S_1$ | $S_1$ | END | $S_6$ | $S_6$ |
| $S_2$ | $S_2$ | $S_2$ | $S_2$ | $S_2$ | $S_2$ | $S_2$ | END |
| $S_3$ | $S_3$ | $S_3$ | $S_3$ | END | $S_5$ | $S_5$ | $S_5$ |
| $S_4$ | $S_4$ | $S_4$ | $S_4$ | $S_4$ | $S_4$ | END | $S_7$ |

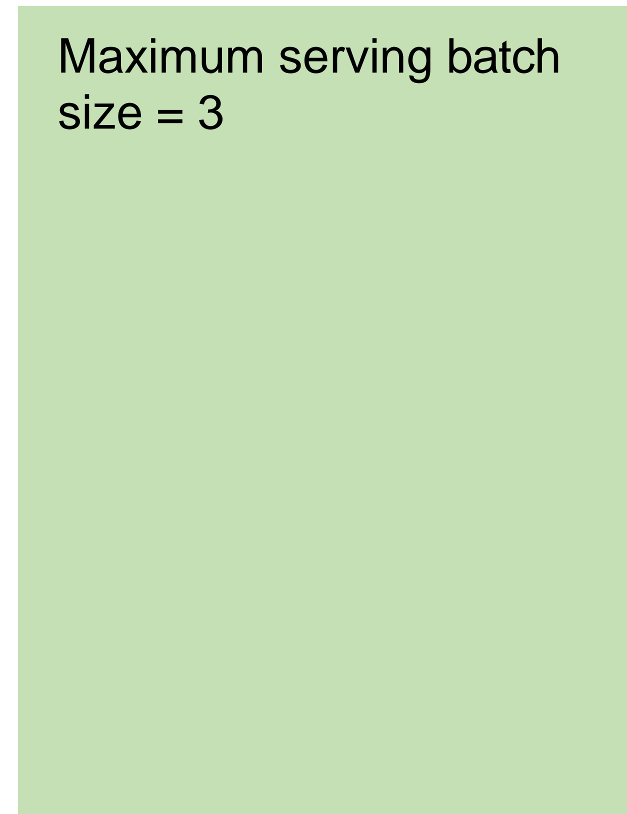
- Higher GPU utilization
- New requests can start immediately

Orca: A Distributed Serving System for Transformer-Based Generative Models. OSDI'22

# Continuous batching step-by-step

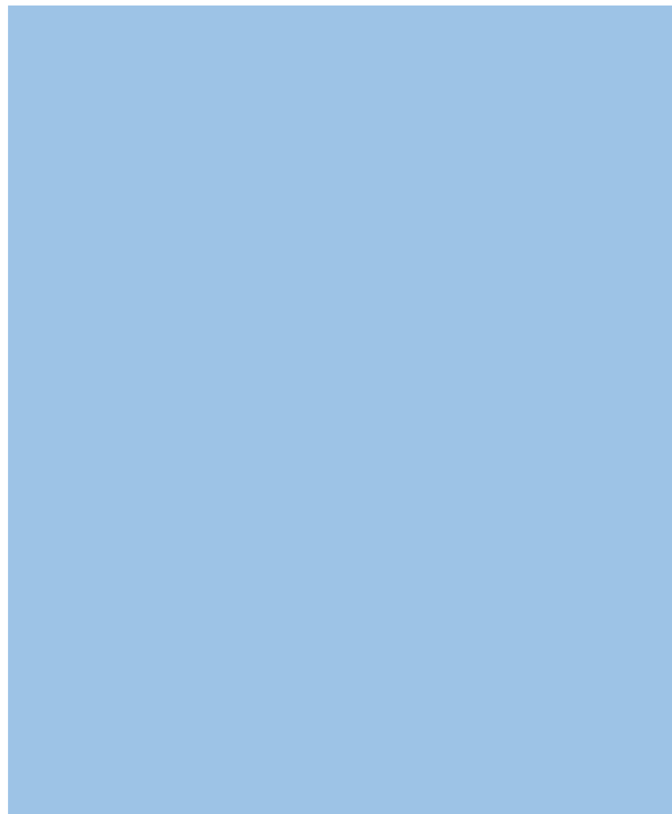
- Receives two new requests R1 and R2

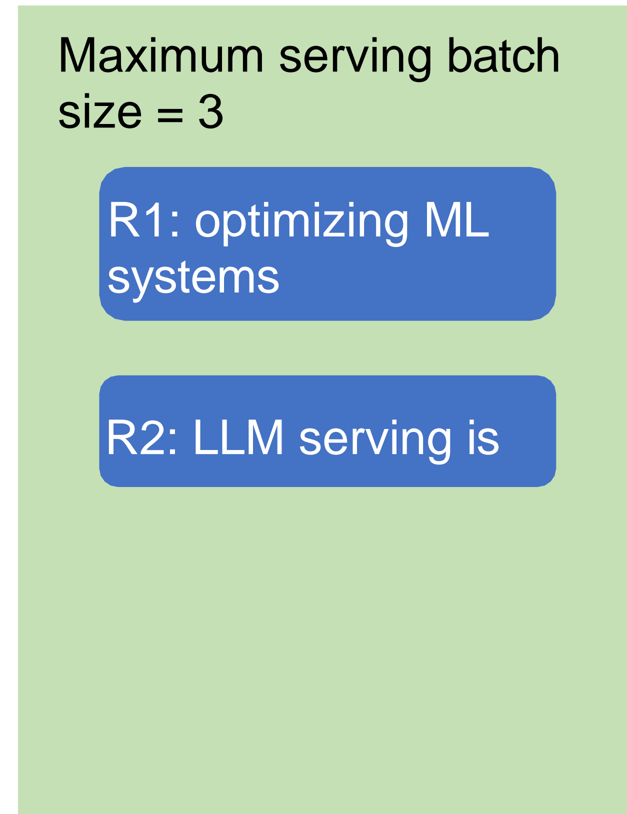


# Continuous batching step-by-step

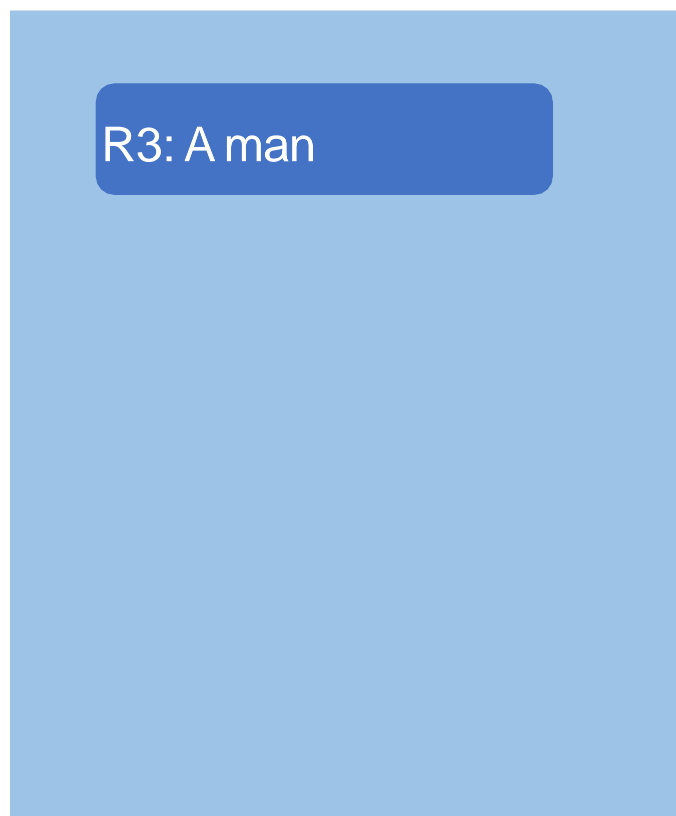
- Iteration 1: decode R1 and R2



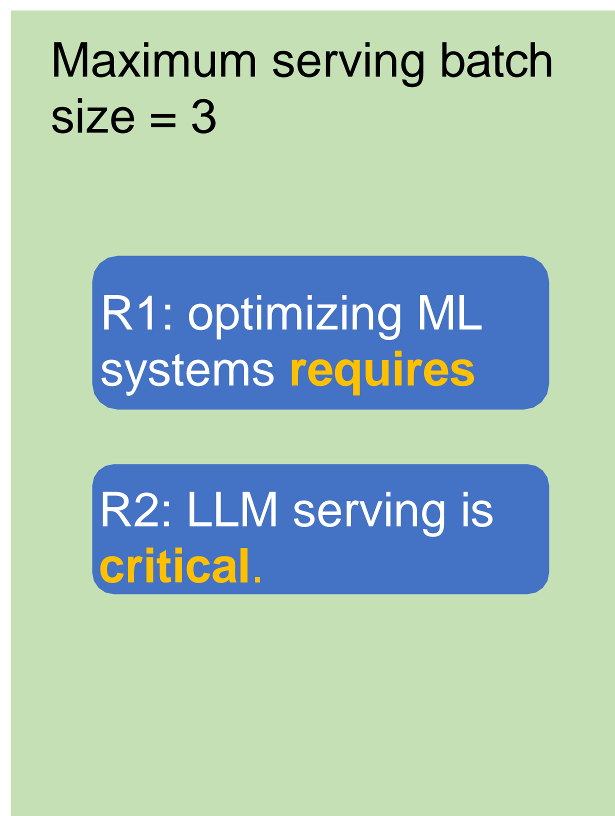



# Continuous batching step-by-step

- Receive a new request R3; finish decoding R1 and R2




Request Pool
(CPU)

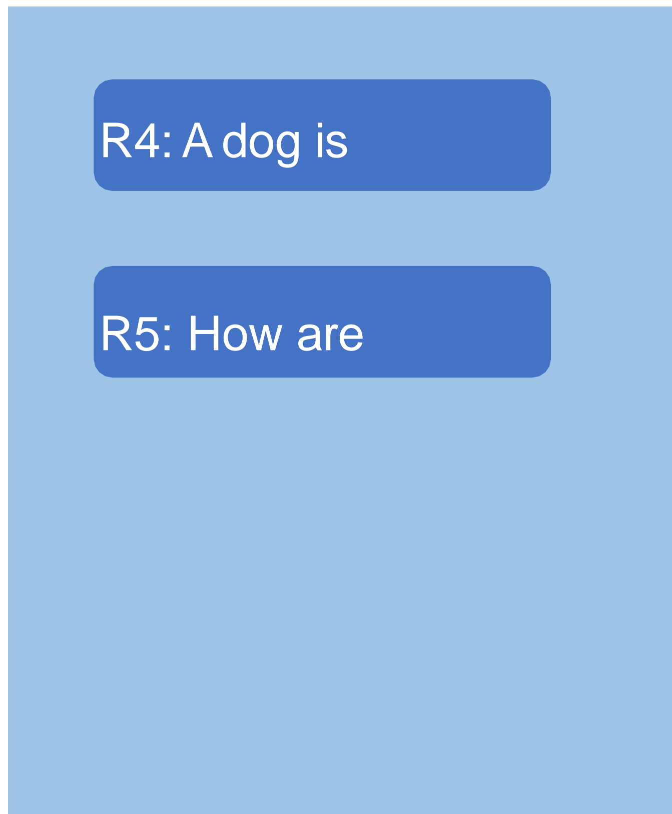



Execution Engine
(GPU)

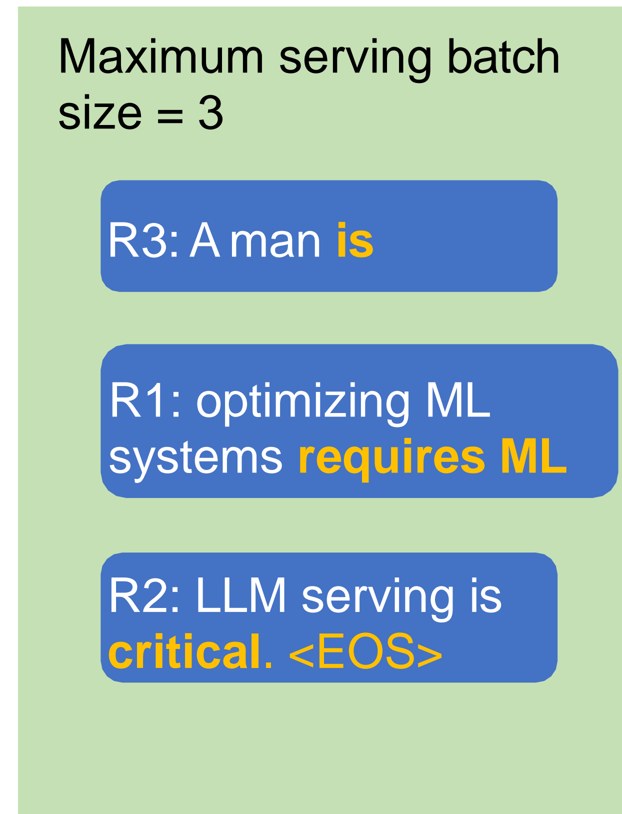
Iteration 1

# Continuous batching step-by-step

- Iteration 2: decode R1, R2, R3; receive R4, R5; R2 completes



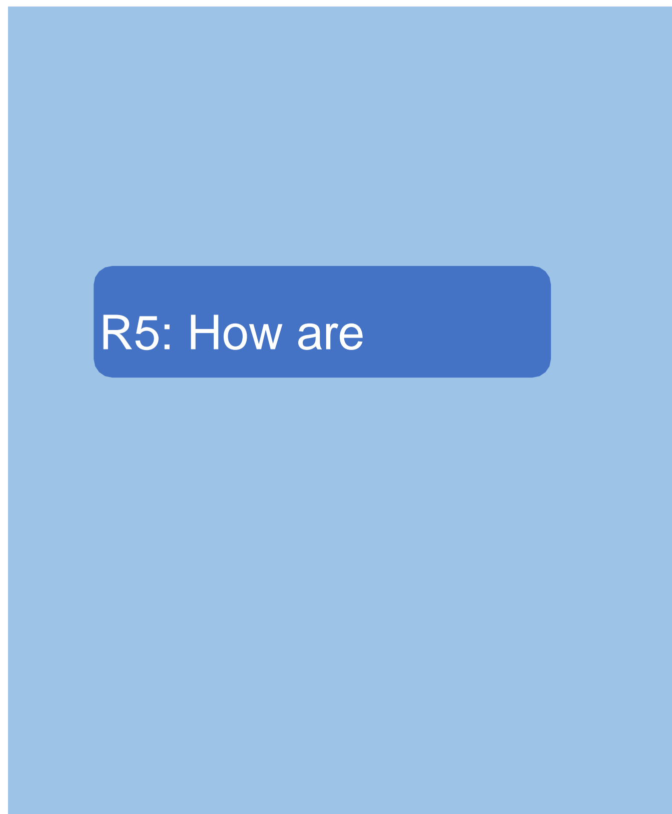

Request Pool (CPU)

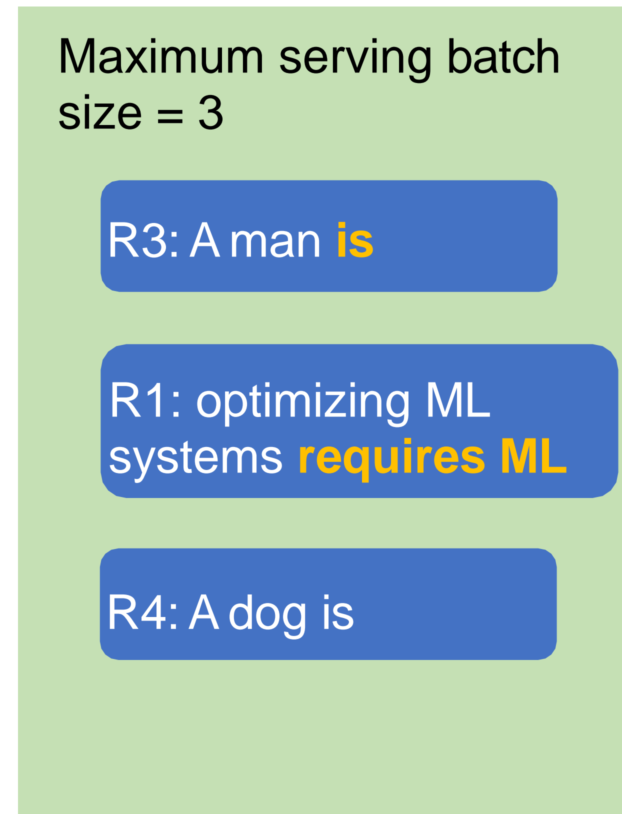



Execution Engine (GPU)

# Continuous batching step-by-step

- Iteration 3: decode R1, R3, R4



Iteration 3

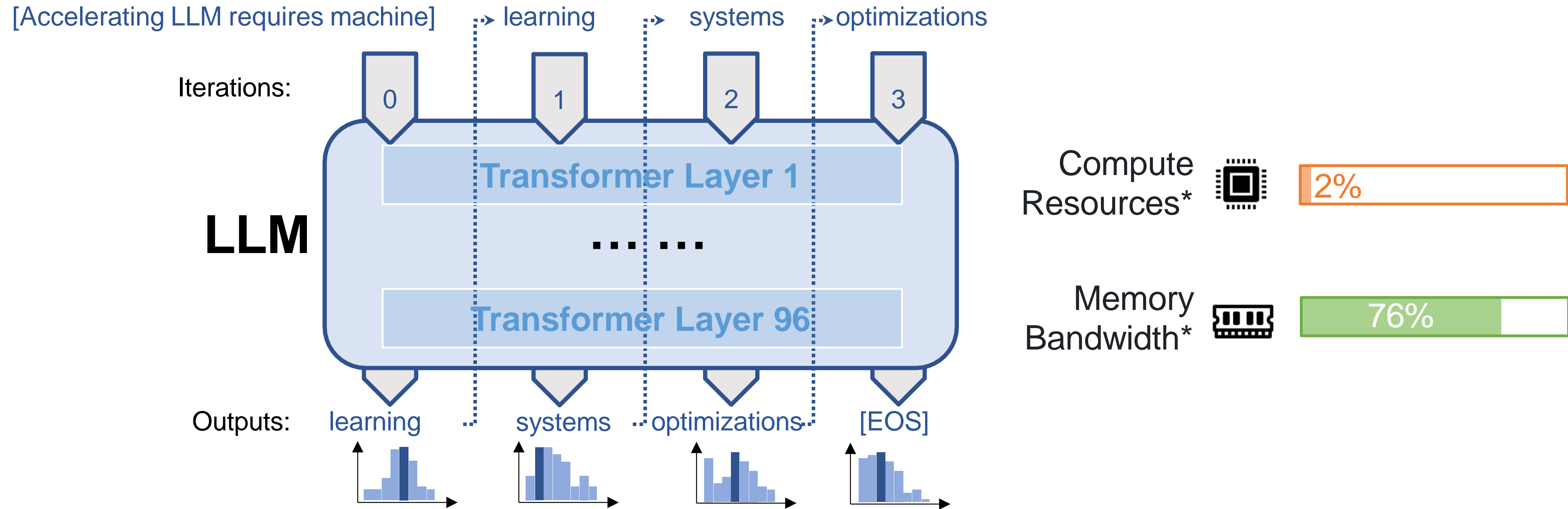
Request Pool (CPU)

Execution Engine (GPU)

# Outline: LLMs serving techniques

- LLM decoding & system design
- Model quantization
- Continuous batching
- Speculative decoding

# Recall: LLM decoding is bottlenecked on memory bandwith




- Limited degree of parallelism → underutilized GPU resources
- Need all parameters to decode a token → bottlenecked by GPU memory access

* Measured by serving LLAMA-2-70B on 4 A100 GPUs with 4K sequence length

# Tradeoffs between language models

| # Parameters | 175B | 13B | 2.7B | 760M | 125M |
|---|---|---|---|---|---|
| TriviaQA | **71.2** | 57.5 | 42.3 | 26.5 | 6.96 |
| PIQA | **82.3** | 79.9 | 75.4 | 72.0 | 64.3 |
| SQuAD | **64.9** | 62.6 | 50.0 | 39.2 | 27.5 |
| latency | 20 s | 7.6s | 2.7s | 1.1s | **0.3s** |
| # A100s | 10 | 1 | 1 | 1 | **1** |

Comparing multiple GPT-3 models*

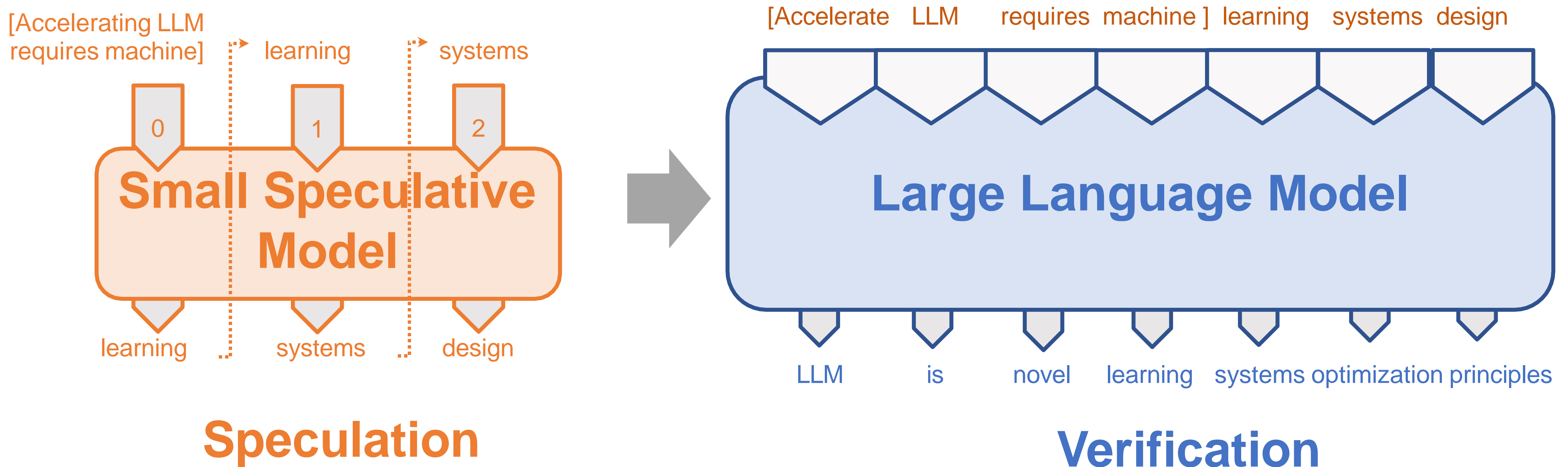
Large models

Small models

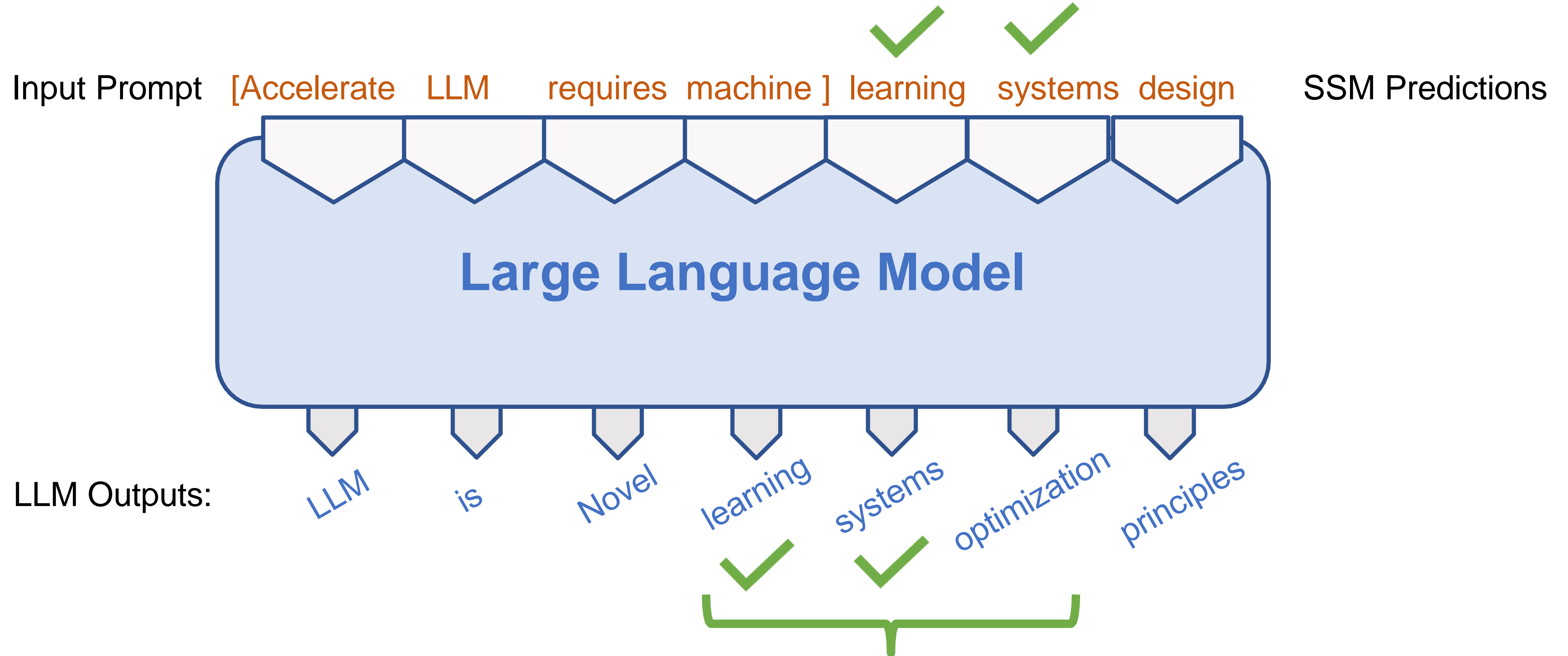
* Language Models are Few-Shot Learners. Arxiv. 2005.14165

# Speculative decoding

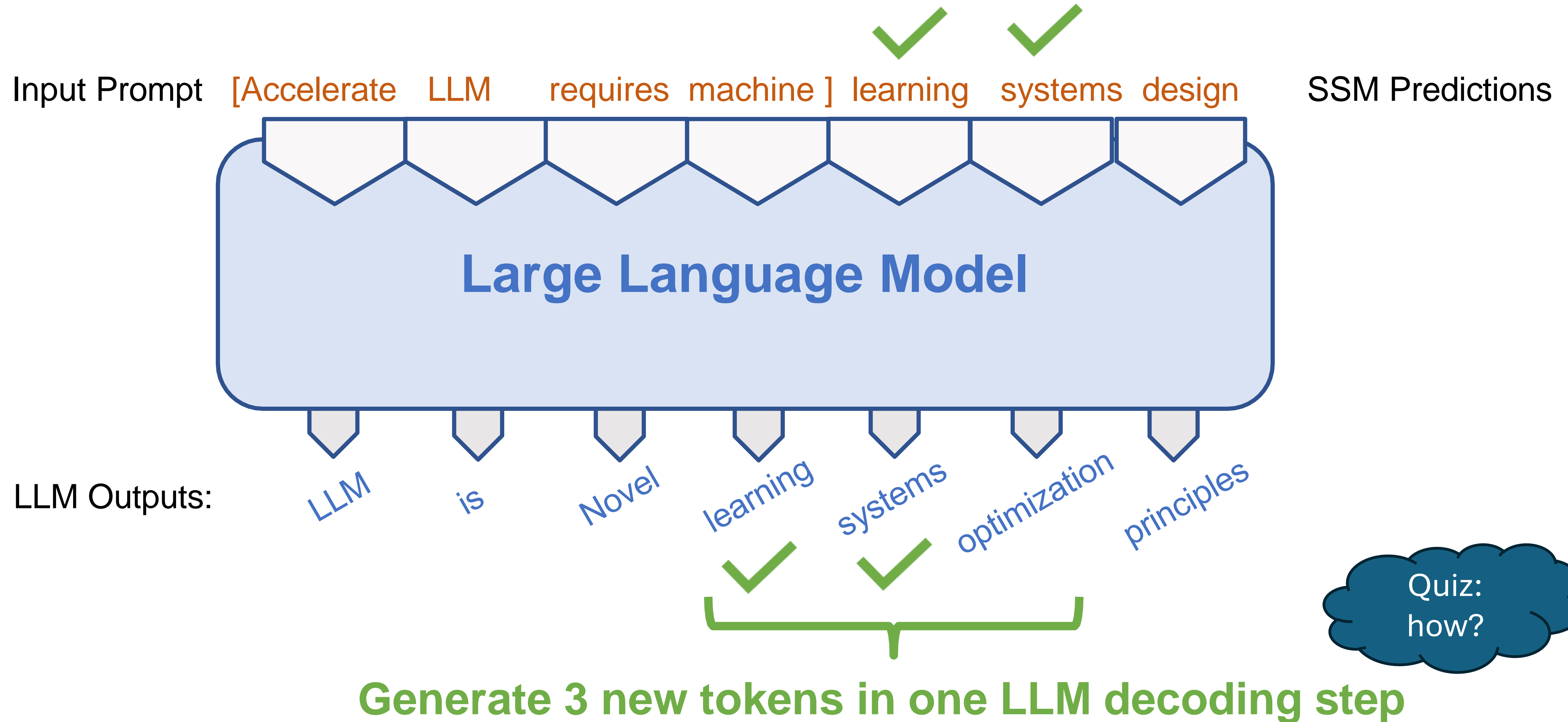
1. Use a small speculative model (SSM) to predict the LLM's output
   - SSM runs much faster than LLM
2. Use the LLM to verify the SSM's prediction



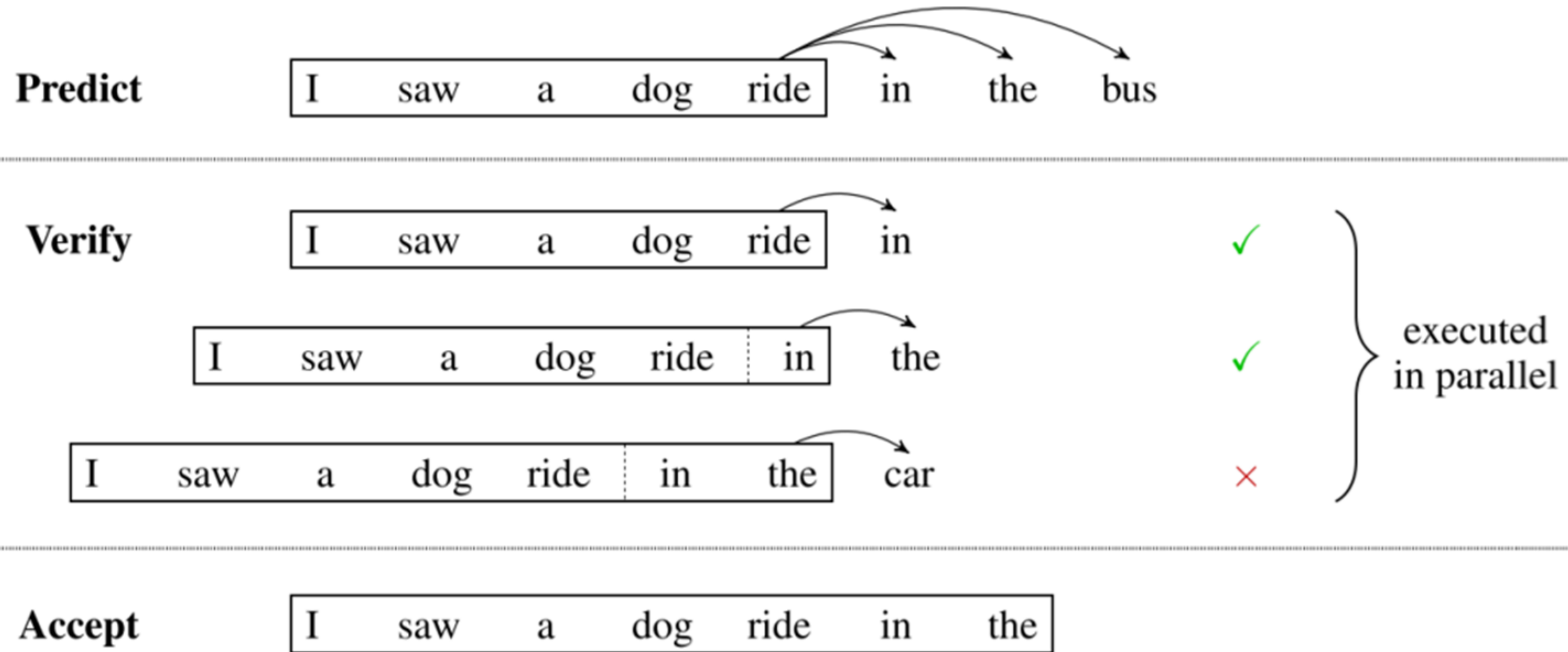
# Verifying speculative decoding results



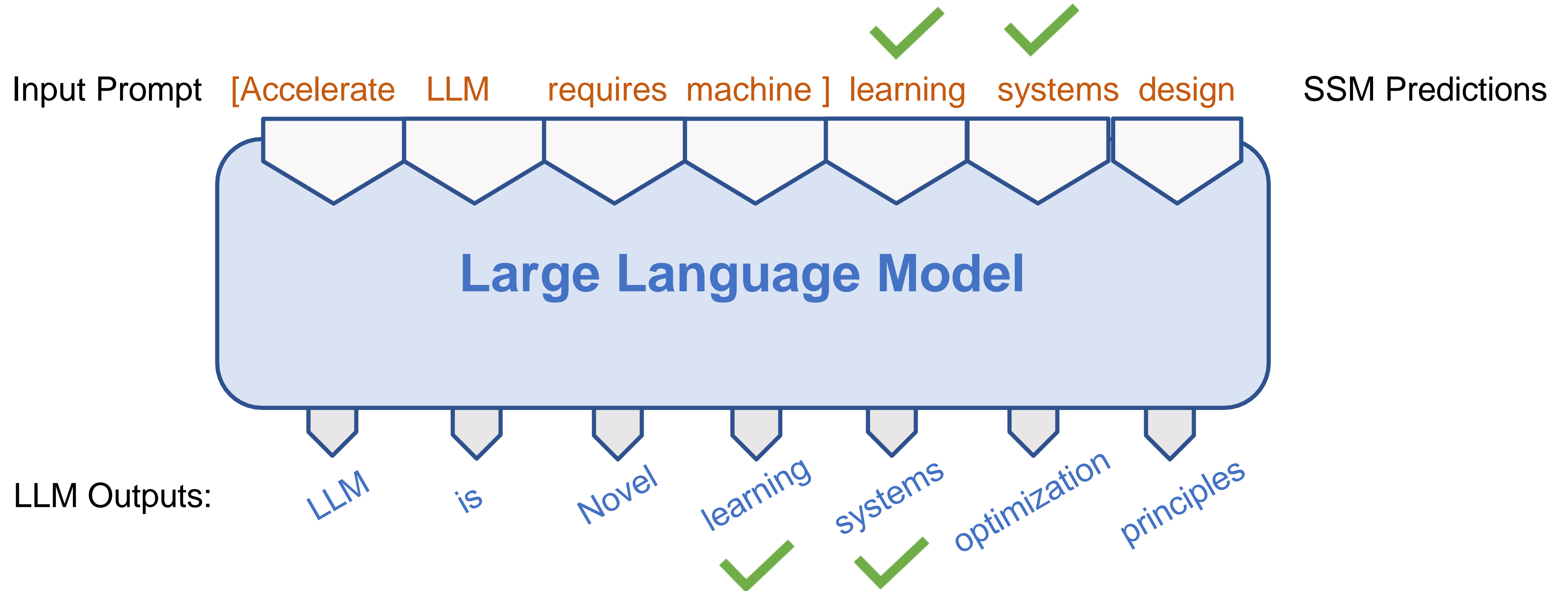
# Verifying speculative decoding results



# Verifying speculative decoding results



# Verifying speculative decoding results




Key takeaway:

- LLM inference is bottlenecked by accessing model weights
- Using LLM to decode multiple tokens to improve GPU utilization
- Tradeoff between latency and throughput

```
================================================
USER: Hi, could you share a tale about a charming llama that grows Medusa-like h
air and starts its own coffee shop?
ASSISTANT:
```

```
================================================
USER: Hi, could you share a tale about a charming llama that grows Medusa-like h
air and starts its own coffee shop?
ASSISTANT:
```

Without speculative decoding

With speculative decoding

(credits from https://github.com/FasterDecoding/Medusa)

# Summary: LLMs serving techniques

- LLM decoding & system design
- Model quantization
- Continuous batching
- Speculative decoding
- What's uncovered
  - Server design & implementation
  - New hardware
  - Compilers
  - Cloud systems
  - Applications

Chef
(LLM)

Restaurant
(serving systems)

Disney world
(cloud systems)