ORACLE

Galloping Llama Experts

Ben Rozonoyer, mentored by Alex Kogan (+ Jeff Diamond)

September 3, 2025

ORACLE

Galloping Llama Experts

Accelerating Llama 4 Token Generation on 1 H100

Ben Rozonoyer, mentored by Alex Kogan (+ Jeff Diamond)

September 3, 2025

ORACLE

Galloping Llama Experts (GLEe)

Accelerating Llama 4 Token Generation on 1 H100

Ben Rozonoyer, mentored by Alex Kogan (+ Jeff Diamond)

September 3, 2025

Intro: What is a Mixture-of-Experts (MoE) Model?

Intro: What is a Mixture-of-Experts (MoE) Model?

From *Adaptive Mixtures of Local Experts* (1991):

- "a new supervised learning procedure for **systems composed of many separate networks, each of which learns to handle a subset of the complete set of training cases**. The new procedure can be viewed either as a **modular version of a multilayer supervised network**, or as an **associative version of competitive learning**."

Intro: What is a Mixture-of-Experts (MoE) Model?

From *Adaptive Mixtures of Local Experts* (1991):

- "a new supervised learning procedure for **systems composed of many separate networks, each of which learns to handle a subset of the complete set of training cases**. The new procedure can be viewed either as a **modular version of a multilayer supervised network**, or as an **associative version of competitive learning**."

From *The Sparsely-Gated Mixture-of-Experts Layer* (2017) — LSTM MoE:

- The capacity of a neural network to absorb information is limited by its number of parameters. **Conditional computation**, where parts of the network are active on a per-example basis, has been proposed in theory as a way of dramatically increasing model capacity without a proportional increase in computation… We introduce a Sparsely-Gated Mixture-of-Experts layer (MoE), consisting of up to thousands of feed-forward sub-networks. **A trainable gating network determines a sparse combination of these experts to use for each example**.

Intro: What is a Mixture-of-Experts (MoE) Model?

From *Adaptive Mixtures of Local Experts* (1991):

- "a new supervised learning procedure for **systems composed of many separate networks, each of which learns to handle a subset of the complete set of training cases**. The new procedure can be viewed either as a **modular version of a multilayer supervised network**, or as an **associative version of competitive learning**."

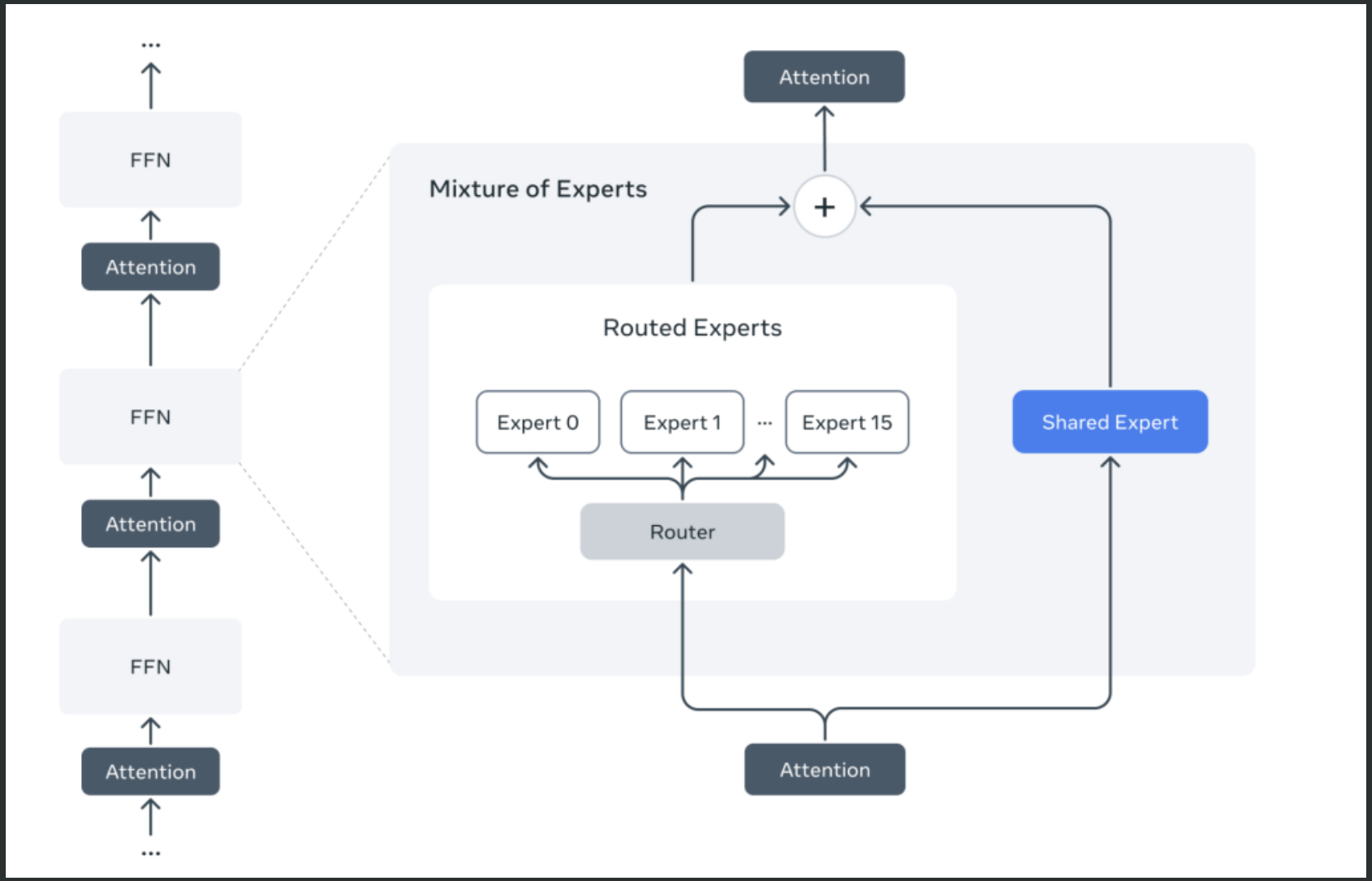
From *The Sparsely-Gated Mixture-of-Experts Layer* (2017) — LSTM MoE:

- The capacity of a neural network to absorb information is limited by its number of parameters. **Conditional computation**, where parts of the network are active on a per-example basis, has been proposed in theory as a way of dramatically increasing model capacity without a proportional increase in computation… We introduce a Sparsely-Gated Mixture-of-Experts layer (MoE), consisting of up to thousands of feed-forward sub-networks. **A trainable gating network determines a sparse combination of these experts to use for each example**.

From *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity* (2021):

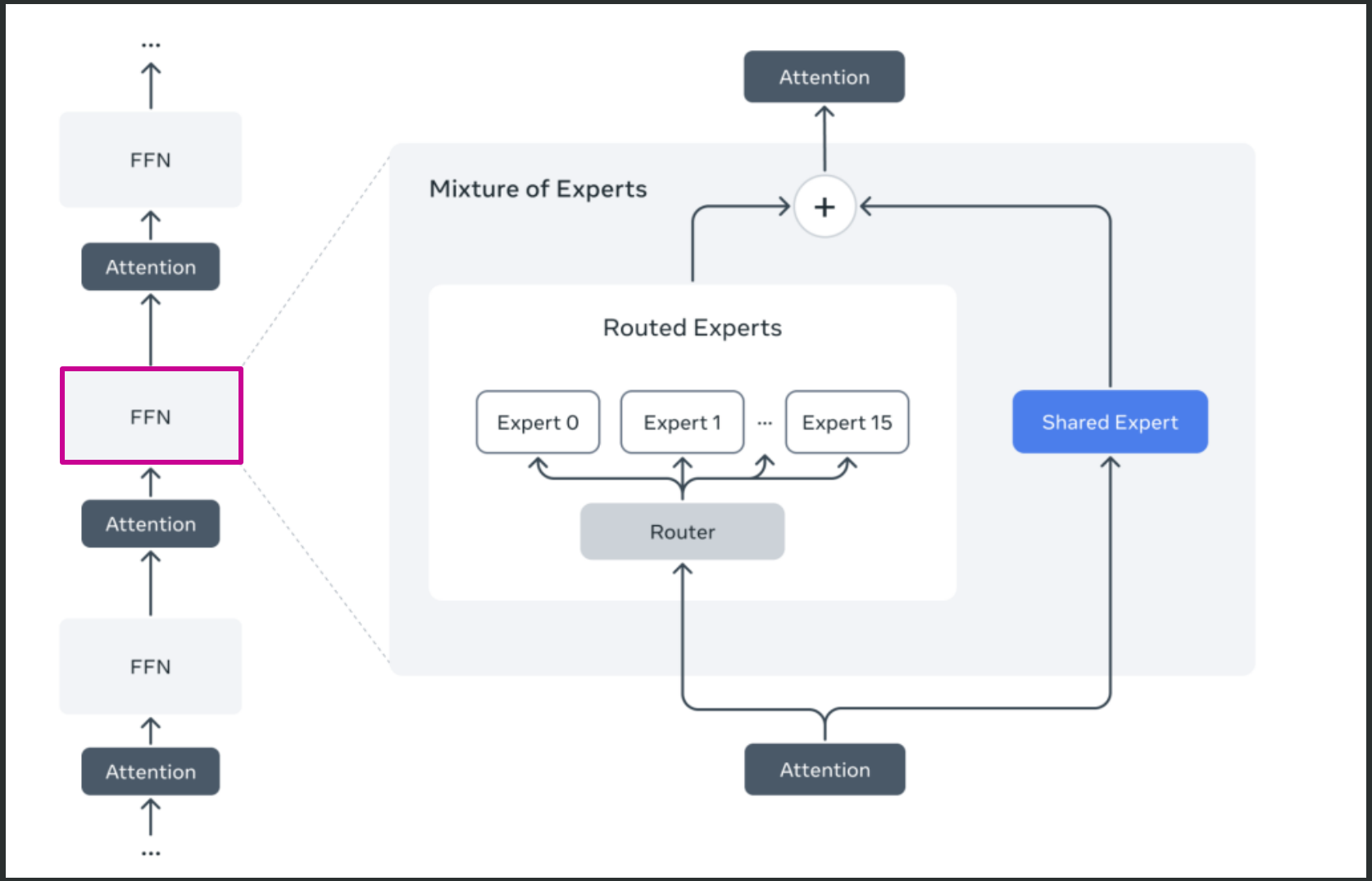
- In deep learning, models typically reuse the same parameters for all inputs. Mixture of Experts (MoE) defies this and instead selects different parameters for each incoming example. The result is a sparsely-activated model -- with **outrageous numbers of parameters -- but a constant computational cost**.

Transformer MoE Illustration



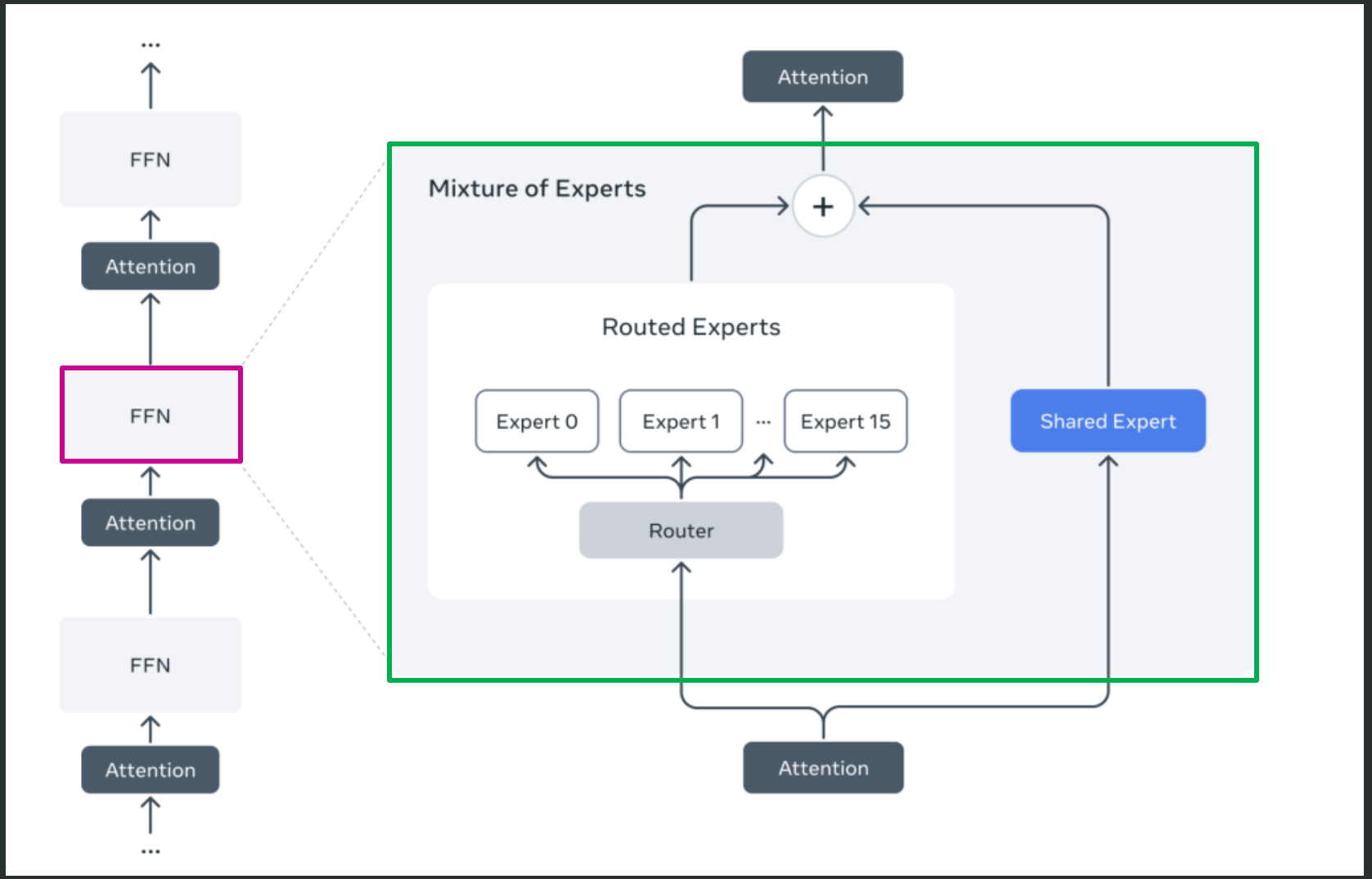
Transformer MoE Illustration

Happens inside each FFN layer



Transformer MoE Illustration

Happens inside each FFN layer



*Activations routed to subset of specialized experts **per token**, and optionally go through a shared expert*

Specific Transformer MoEs

SwitchTransformers

DBRX

Mixtral

Sparsely-Gated MoE

OLMoE

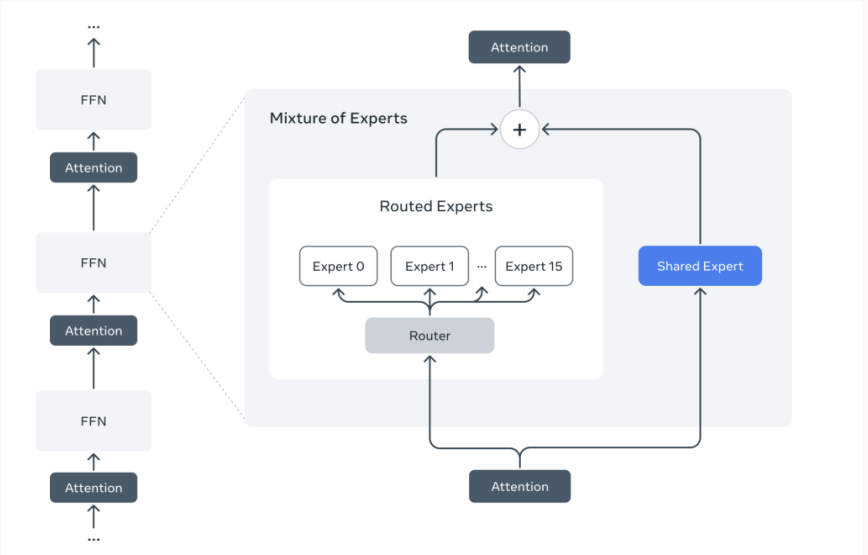
Qwen

Gemini 1.5

DeepSeek (MoE, V2, V3)

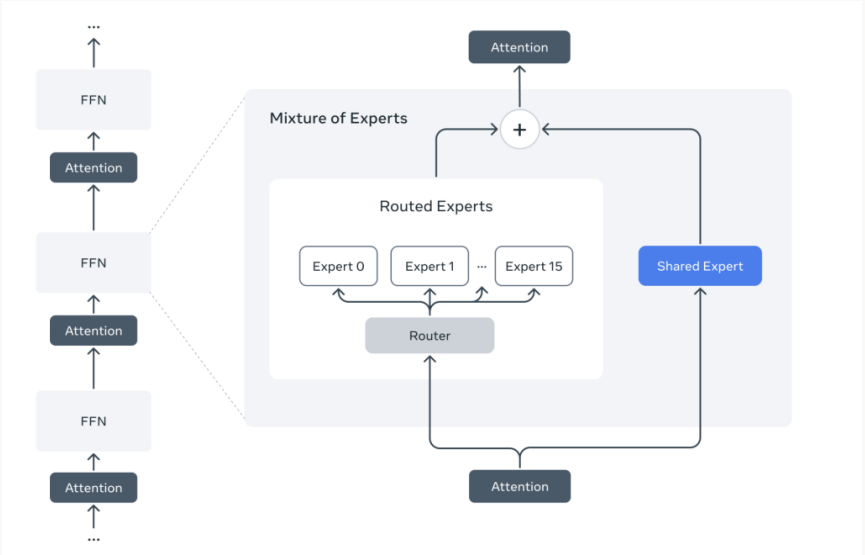
Llama 4 (Scout, Maverick, Behemoth)

Llama 4 Scout: Expert Parameters Info



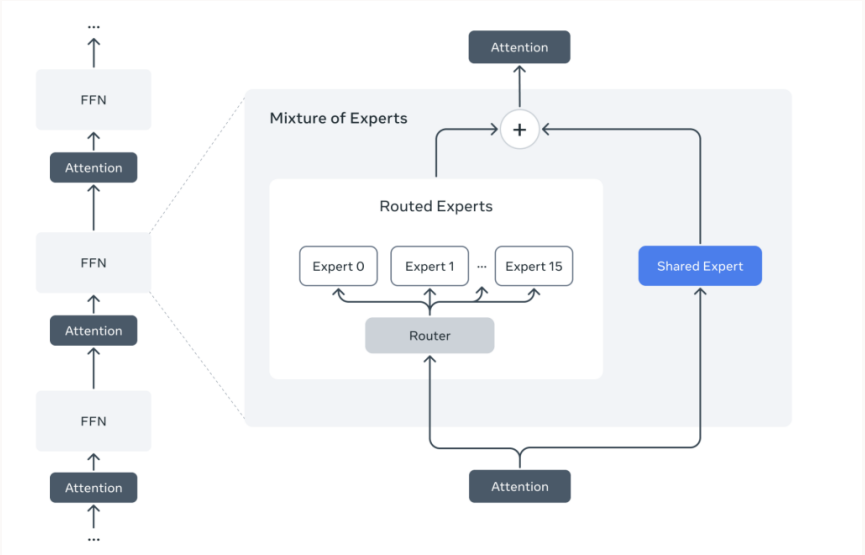
Llama 4 Scout: Expert Parameters Info

- Llama 4 Scout:
  - 17B active parameters
  - 109B total parameters
  - 48 layers
  - 16 experts / layer, 1 gets activated



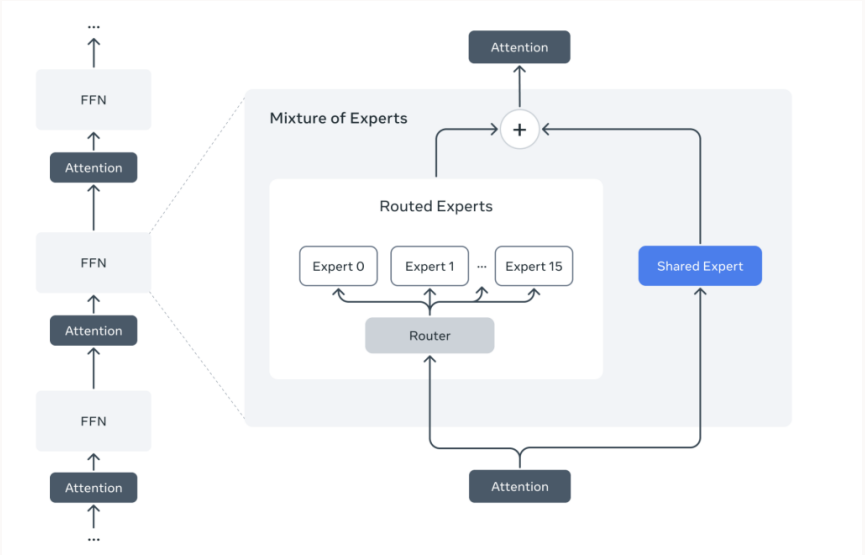
Llama 4 Scout: Expert Parameters Info

- Llama 4 Scout:
  - 17B active parameters
  - 109B total parameters
  - 48 layers
  - 16 experts / layer, 1 gets activated



- Each [SwiGLU] expert consists of the following 2 (logically 3) bfloat16 weights:
  - **w13_weight**: 16384 x 5120
  - **w2_weight**: 5120 x 8192

Llama 4 Scout: Expert Parameters Info

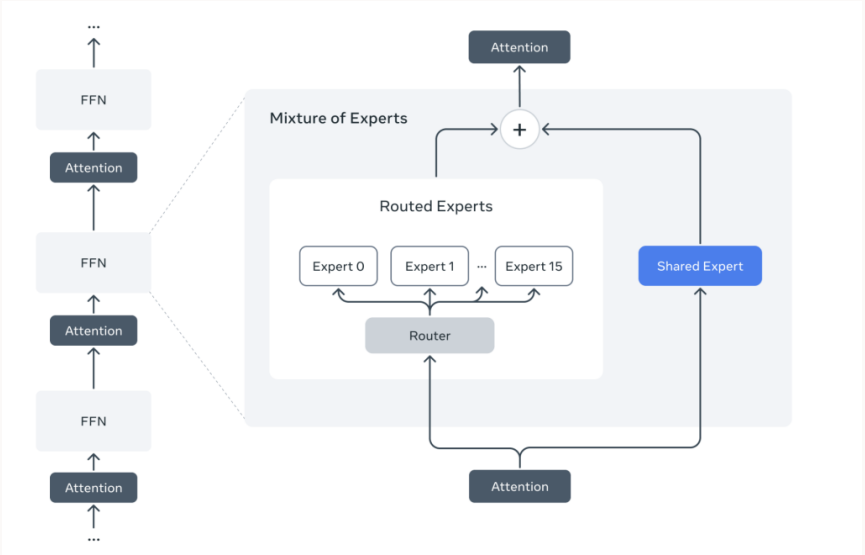


- Llama 4 Scout:
  - 17B active parameters
  - 109B total parameters
  - 48 layers
  - 16 experts / layer, 1 gets activated

- Each [SwiGLU] expert consists of the following 2 (logically 3) bfloat16 weights:
  - **w13_weight**: 16384 x 5120
  - **w2_weight**: 5120 x 8192

- In total, this amounts to:
  - $(16384 * 5120 + 5120 * 8192)_{\text{params/expert}} * 2_{\text{bytes/bfloat16}}$
  - $= 251{,}658{,}240_{\text{bytes/expert}}$ ≅ **0.25 GB per expert**
  - $* 16_{\text{experts/layer}}$ ≅ **4 GB per Llama4DecoderLayer**
  - $* 48_{\text{layers/model}}$ ≅ **192 GB in Llama 4 Scout**

Llama 4 Scout: Expert Parameters Info



- Llama 4 Scout:
  - 17B active parameters
  - 109B total parameters
  - 48 layers
  - 16 experts / layer, 1 gets activated

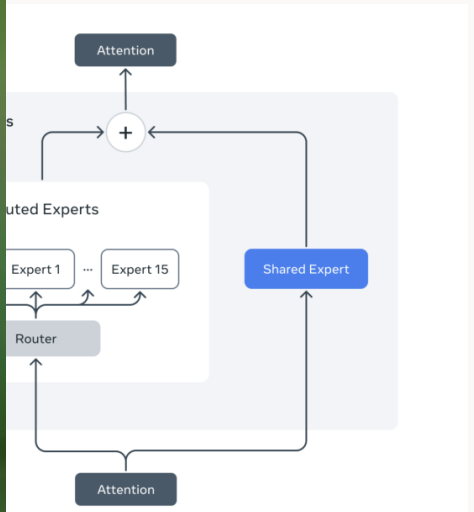
- Each [SwiGLU] expert consists of the following 2 (logically 3) bfloat16 weights:
  - **w13_weight**: 16384 x 5120
  - **w2_weight**: 5120 x 8192

- In total, this amounts to:
  - $(16384 * 5120 + 5120 * 8192)_{\text{params/expert}} * 2_{\text{bytes/bfloat16}}$
  - $= 251{,}658{,}240_{\text{bytes/expert}}$ ≅ **0.25 GB per expert**
  - $* 16_{\text{experts/layer}}$ ≅ **4 GB per Llama4DecoderLayer**
  - $* 48_{\text{layers/model}}$ ≅ **192 GB in Llama 4 Scout**

- **Meanwhile, a single H100 has only 80GB of HBM3 memory… Need large number of GPUs to serve**

Llama 4 Scout:

- Llama 4 Scout:
  - 17B active p
  - 109B total p
  - 48 layers
  - 16 experts /
- Each [SwiGLU] ex
  - **w13_weigh**
  - **w2_weight**
- In total, this amo
  - (16384 * 5
  - = 251,658,2
  - * 16$_{\text{experts/lay}}$
  - * 48$_{\text{layers/mod}}$
- **Meanwhile, a si**



MoE Inference: Concepts, Challenges, Strategies

MoE Inference: Concepts, Challenges, Strategies

- **Prompt Processing (PP), aka Prefill**:
  - compute-bound (many tokens at a time, most of the experts will be required)

MoE Inference: Concepts, Challenges, Strategies

- **Prompt Processing (PP), aka Prefill**:
  - compute-bound (many tokens at a time, most of the experts will be required)
- **Token Generation (TG)**:
  - memory-bound (one token at a time, only top-*k*/top-1 experts required)

MoE Inference: Concepts, Challenges, Strategies

- **Prompt Processing (PP), aka Prefill**:
  - compute-bound (many tokens at a time, most of the experts will be required)
- **Token Generation (TG)**:
  - memory-bound (one token at a time, only top-*k*/top-1 experts required)

- **Challenge #1:**
  - Need access to all MoE experts, but can't fit them all into 1 GPU

MoE Inference: Concepts, Challenges, Strategies

- **Prompt Processing (PP), aka Prefill**:
  - compute-bound (many tokens at a time, most of the experts will be required)
- **Token Generation (TG)**:
  - memory-bound (one token at a time, only top-*k*/top-1 experts required)

- **Challenge #1:**
  - Need access to all MoE experts, but can't fit them all into 1 GPU
- **Strategy #1:**
  - **Caching** — most frequent/important/expected experts on GPU, rest on CPU

MoE Inference: Concepts, Challenges, Strategies

- **Prompt Processing (PP), aka Prefill**:
  - compute-bound (many tokens at a time, most of the experts will be required)
- **Token Generation (TG)**:
  - memory-bound (one token at a time, only top-*k*/top-1 experts required)

- **Challenge #1:**
  - Need access to all MoE experts, but can't fit them all into 1 GPU
- **Strategy #1:**
  - **Caching** — most frequent/important/expected experts on GPU, rest on CPU

- **Challenge #2:**
  - Computing with weights on GPU is much faster than loading from CPU to GPU — pipeline bubbles

MoE Inference: Concepts, Challenges, Strategies

- **Prompt Processing (PP), aka Prefill**:
  - compute-bound (many tokens at a time, most of the experts will be required)
- **Token Generation (TG)**:
  - memory-bound (one token at a time, only top-*k*/top-1 experts required)

- **Challenge #1:**
  - Need access to all MoE experts, but can't fit them all into 1 GPU
- **Strategy #1:**
  - **Caching** — most frequent/important/expected experts on GPU, rest on CPU

- **Challenge #2:**
  - Computing with weights on GPU is much faster than loading from CPU to GPU — pipeline bubbles
- **Strategy #2:**
  - **Copy-compute overlap** — start **prefetching** in advance

Our Task

Improve latency of token generation (TG) phase of Llama 4 Scout inference on 1 NVIDIA H100 GPU, under the single-user scenario with batch size 1.

Our Task

Improve latency of token generation (TG) phase of Llama 4 Scout inference on 1 NVIDIA H100 GPU, under the single-user scenario with batch size 1.

Our Task

Improve latency of token generation (TG) phase of Llama 4 Scout inference on 1 NVIDIA H100 GPU, under the single-user scenario with batch size 1.

Our Task

Improve latency of token generation (TG) phase of Llama 4 Scout inference on 1 NVIDIA H100 GPU, under the single-user scenario with batch size 1.

Our Task

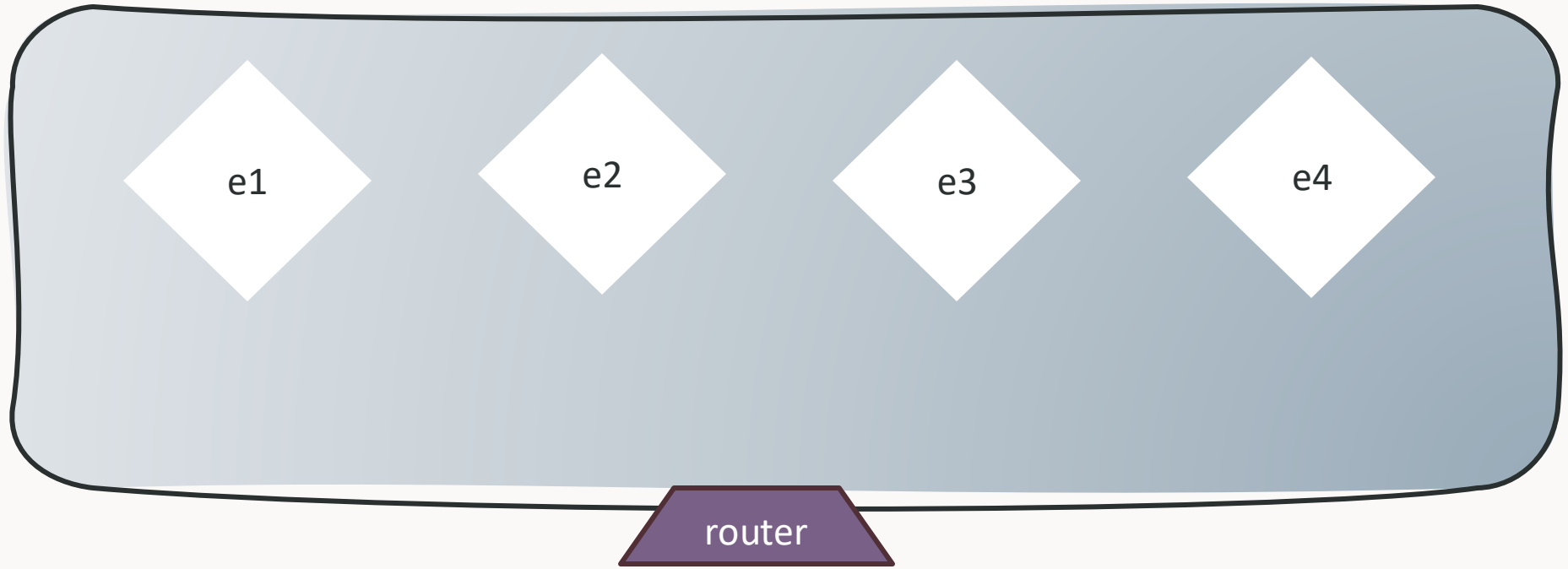
Improve latency of token generation (TG) phase of Llama 4 Scout inference on 1 NVIDIA H100 GPU, under the single-user scenario with batch size 1.

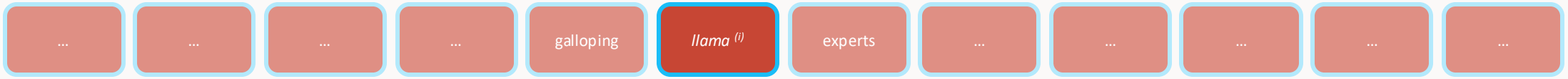
Our Task

Improve latency of token generation (TG) phase of Llama 4 Scout inference on 1 NVIDIA H100 GPU, under the single-user scenario with batch size 1.

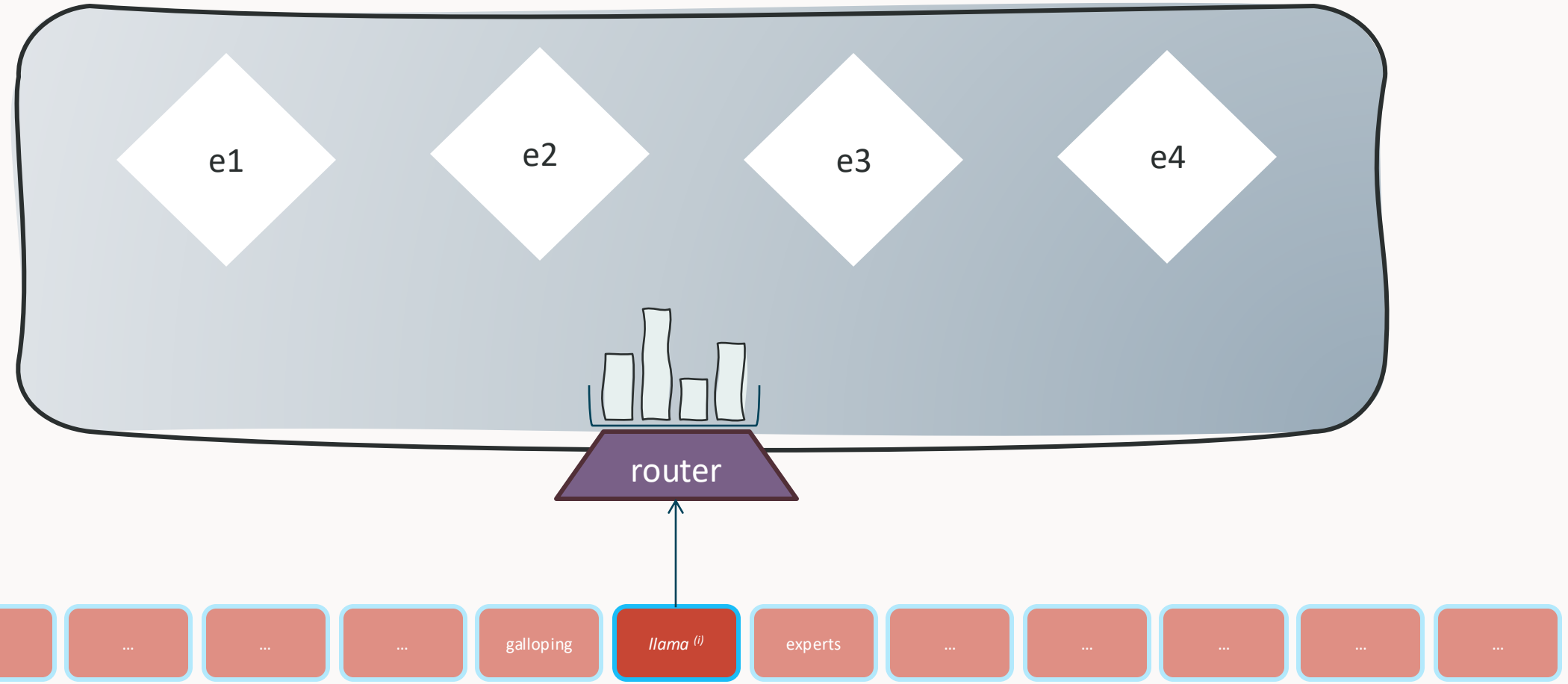
MoE + Prefetching Illustration

Llama4MoE Layer

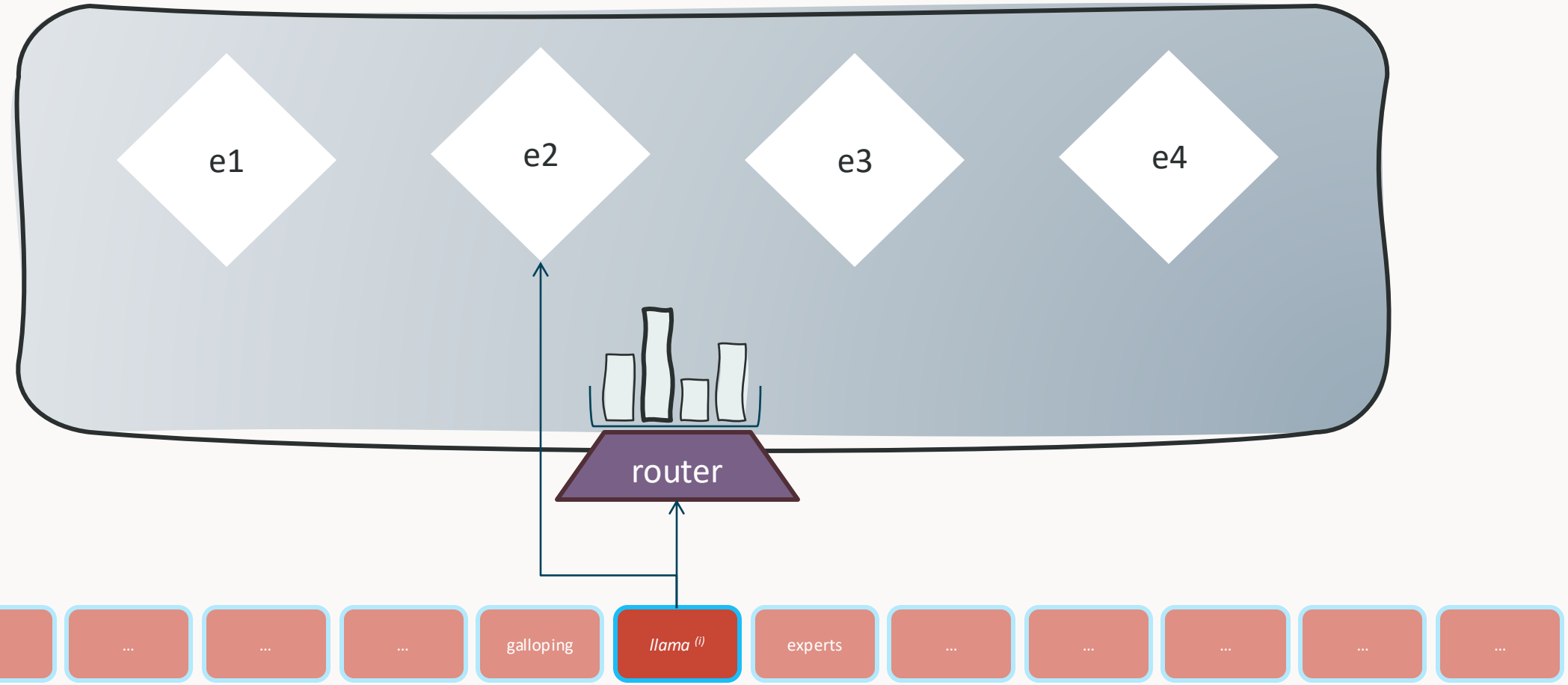




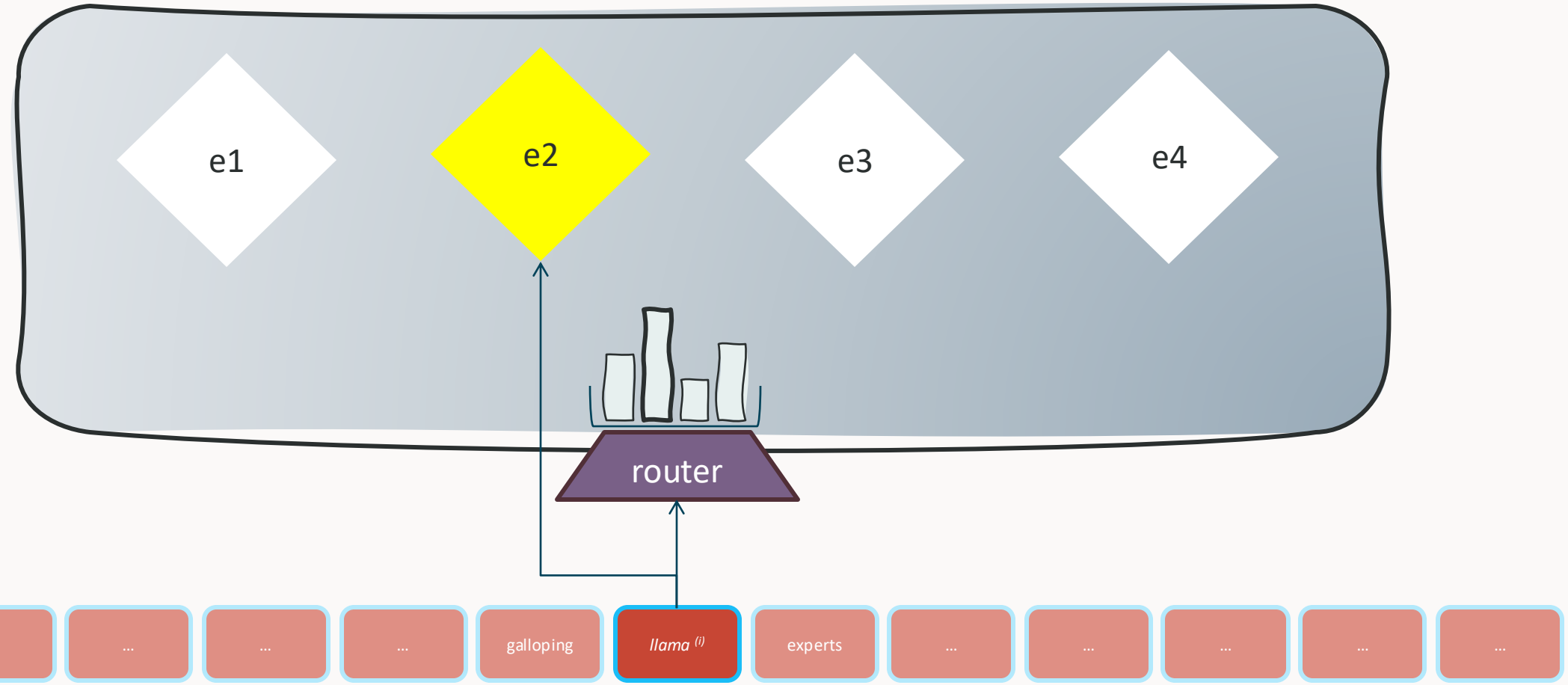
Llama4MoE Layer



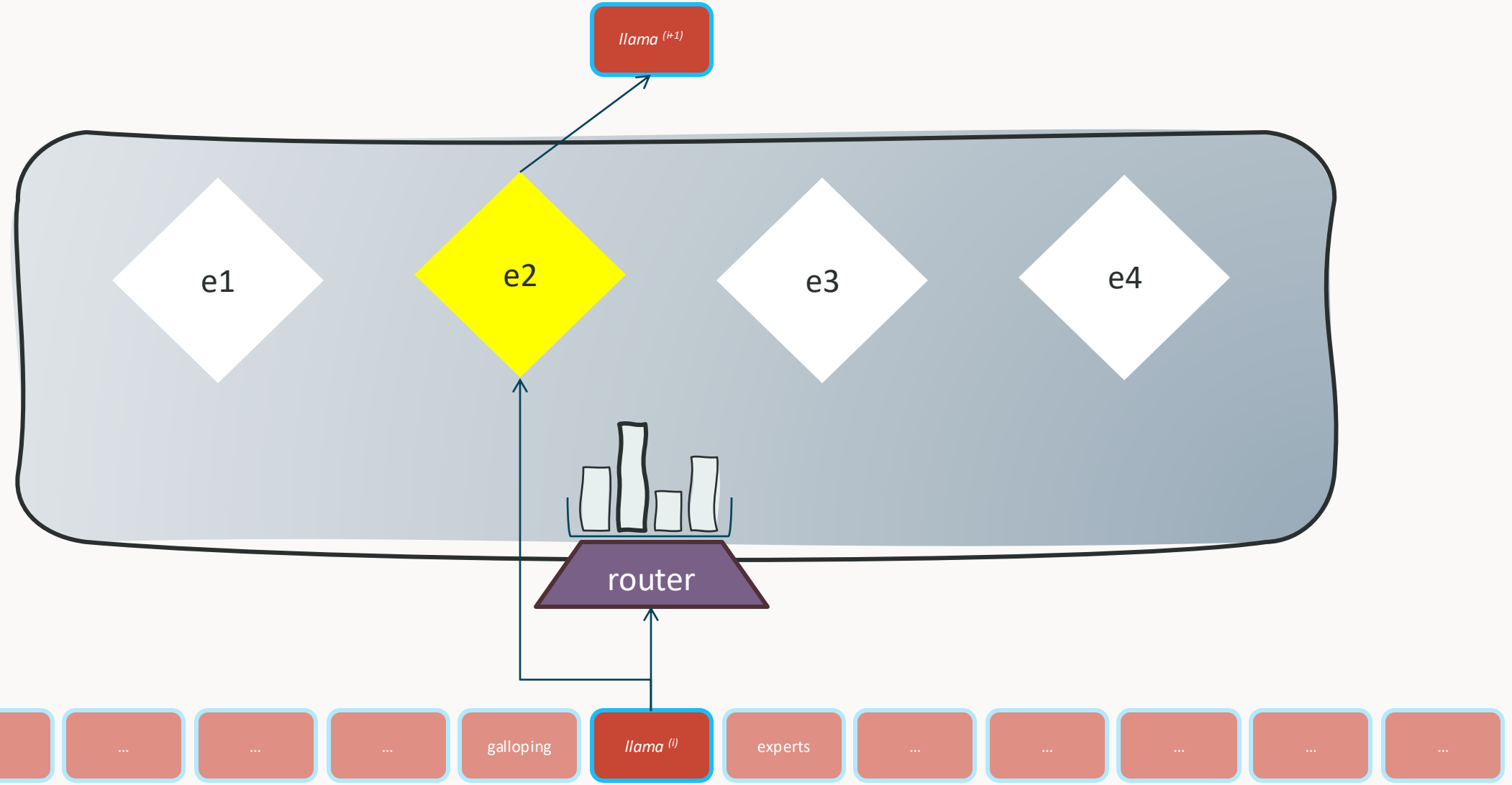
Llama4MoE Layer

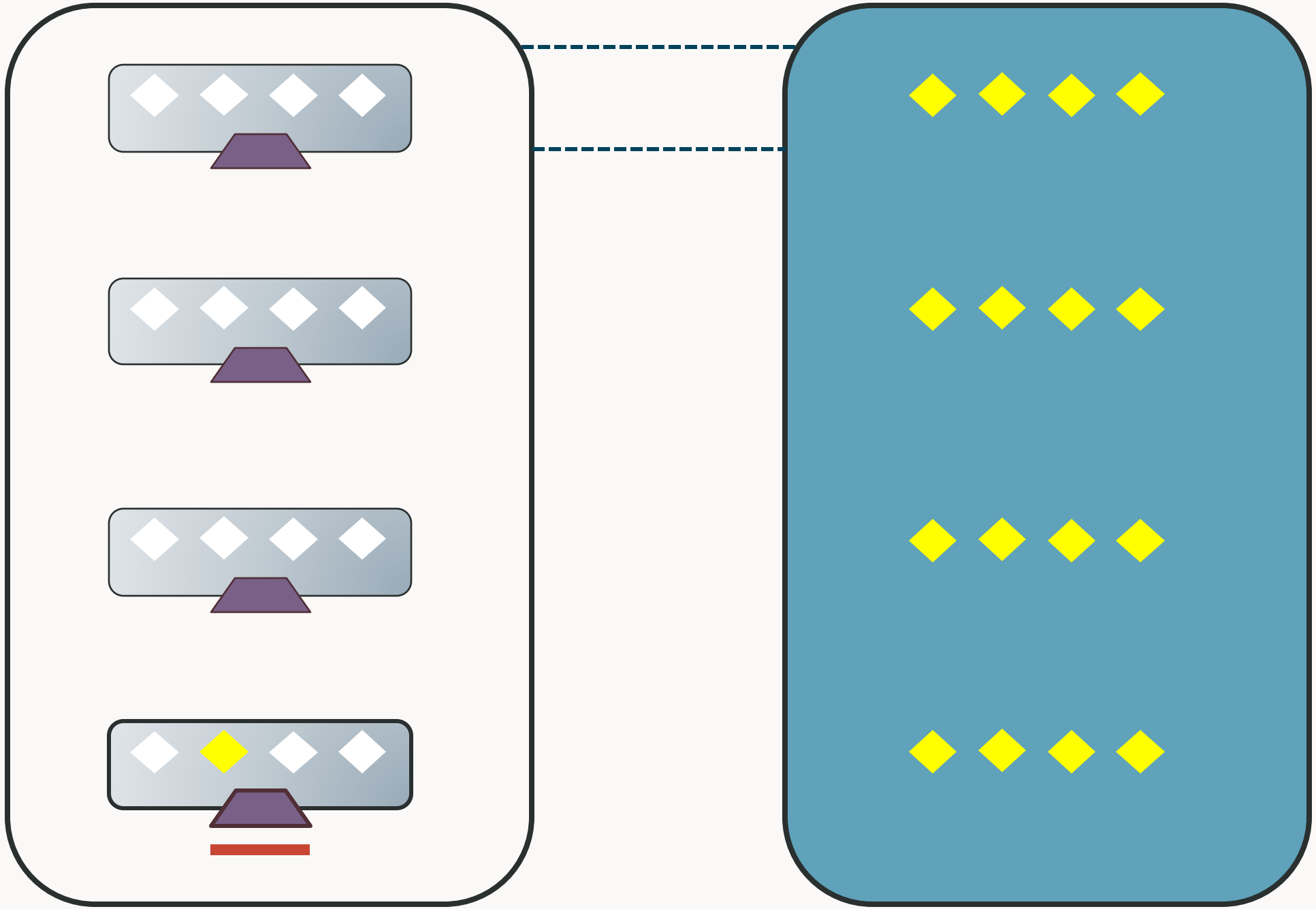


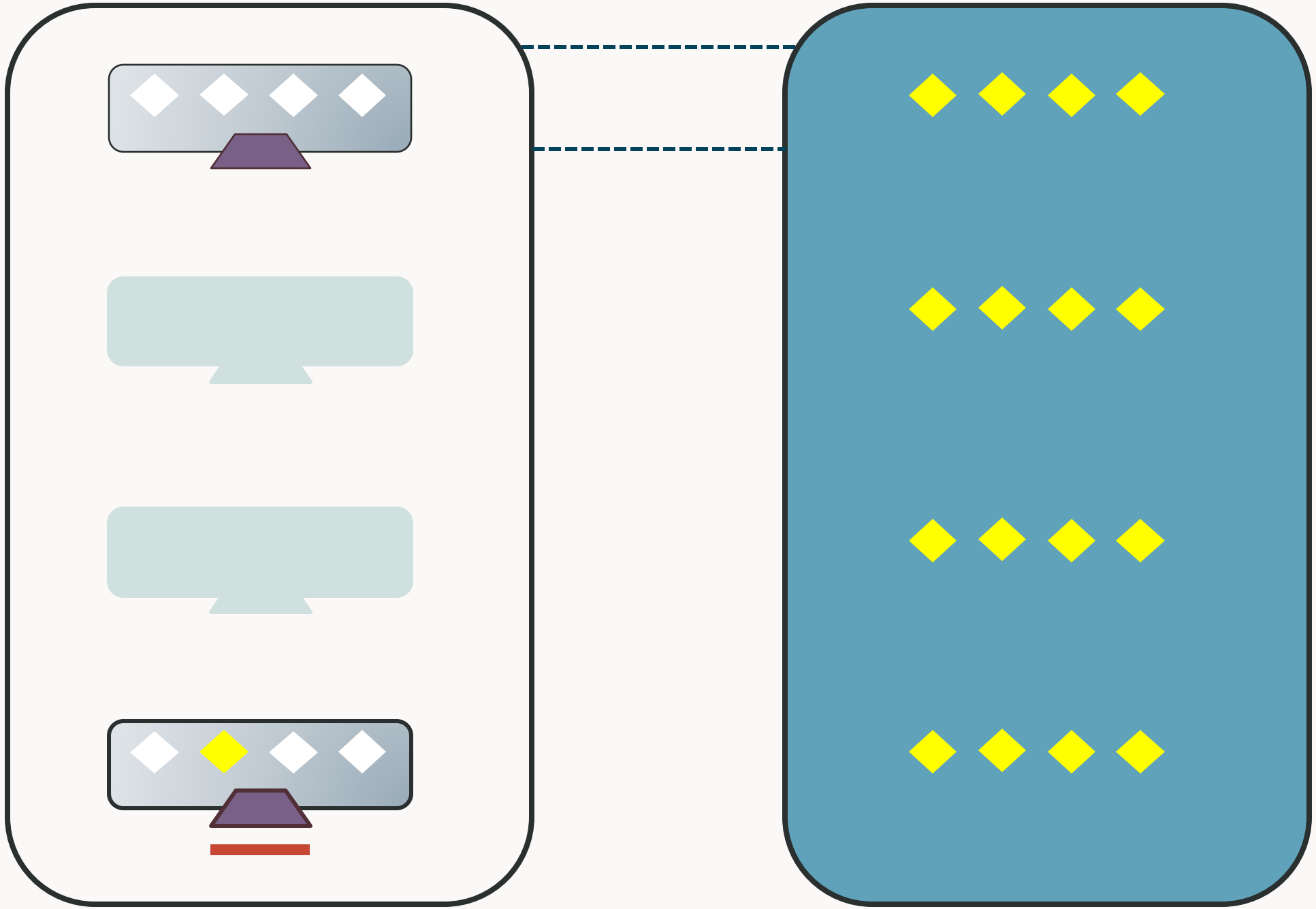
Llama4MoE Layer

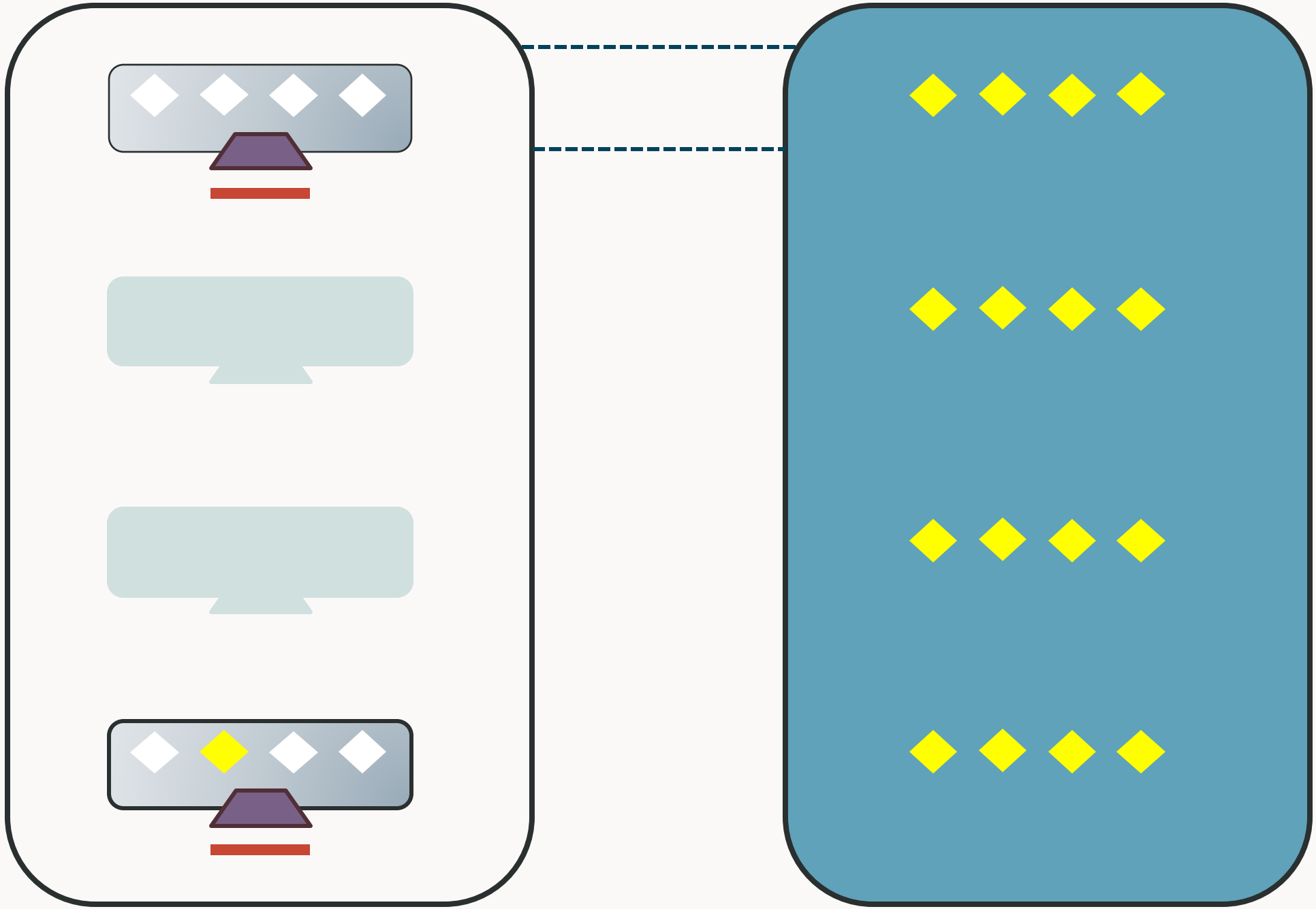


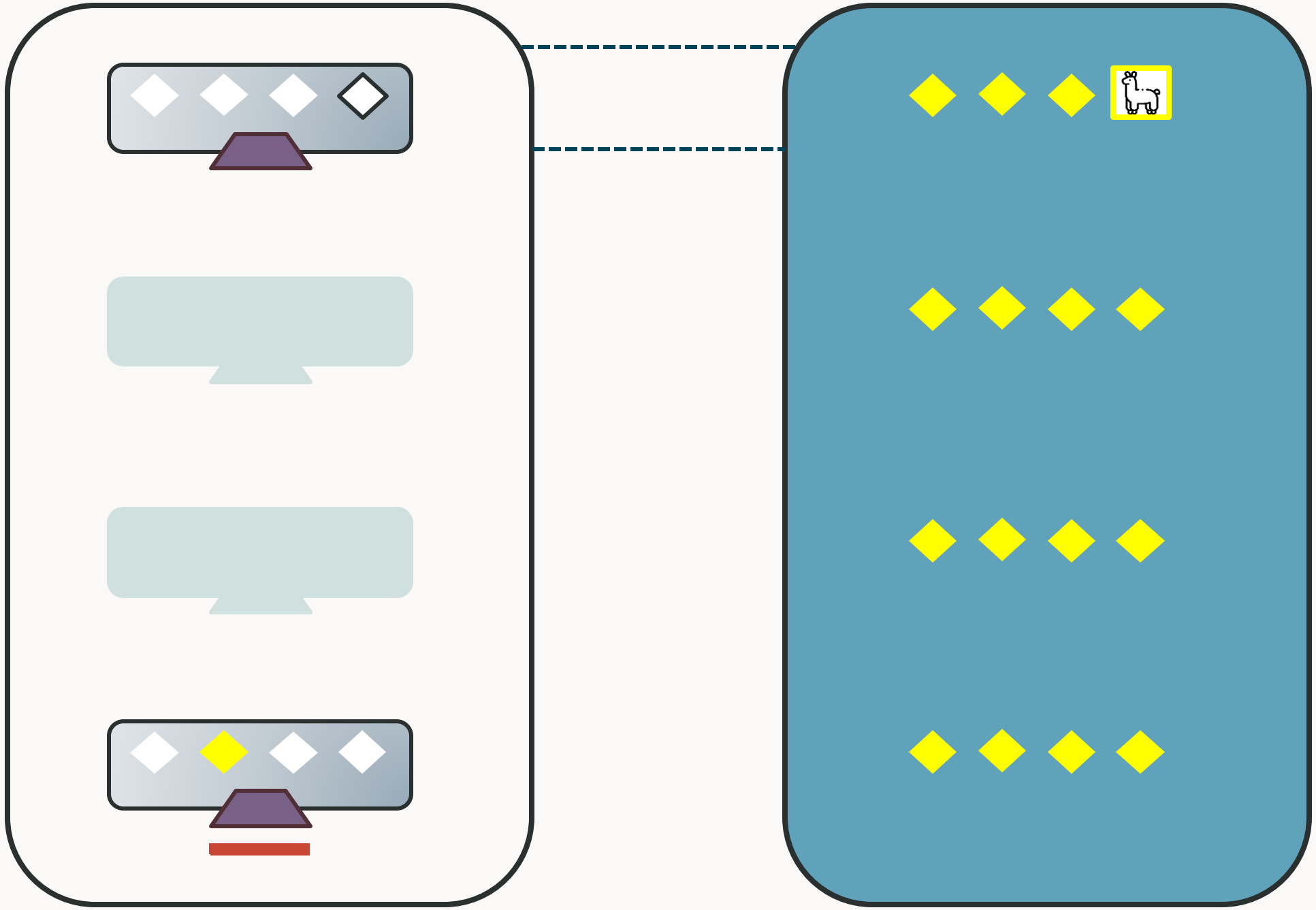
Llama4MoE Layer

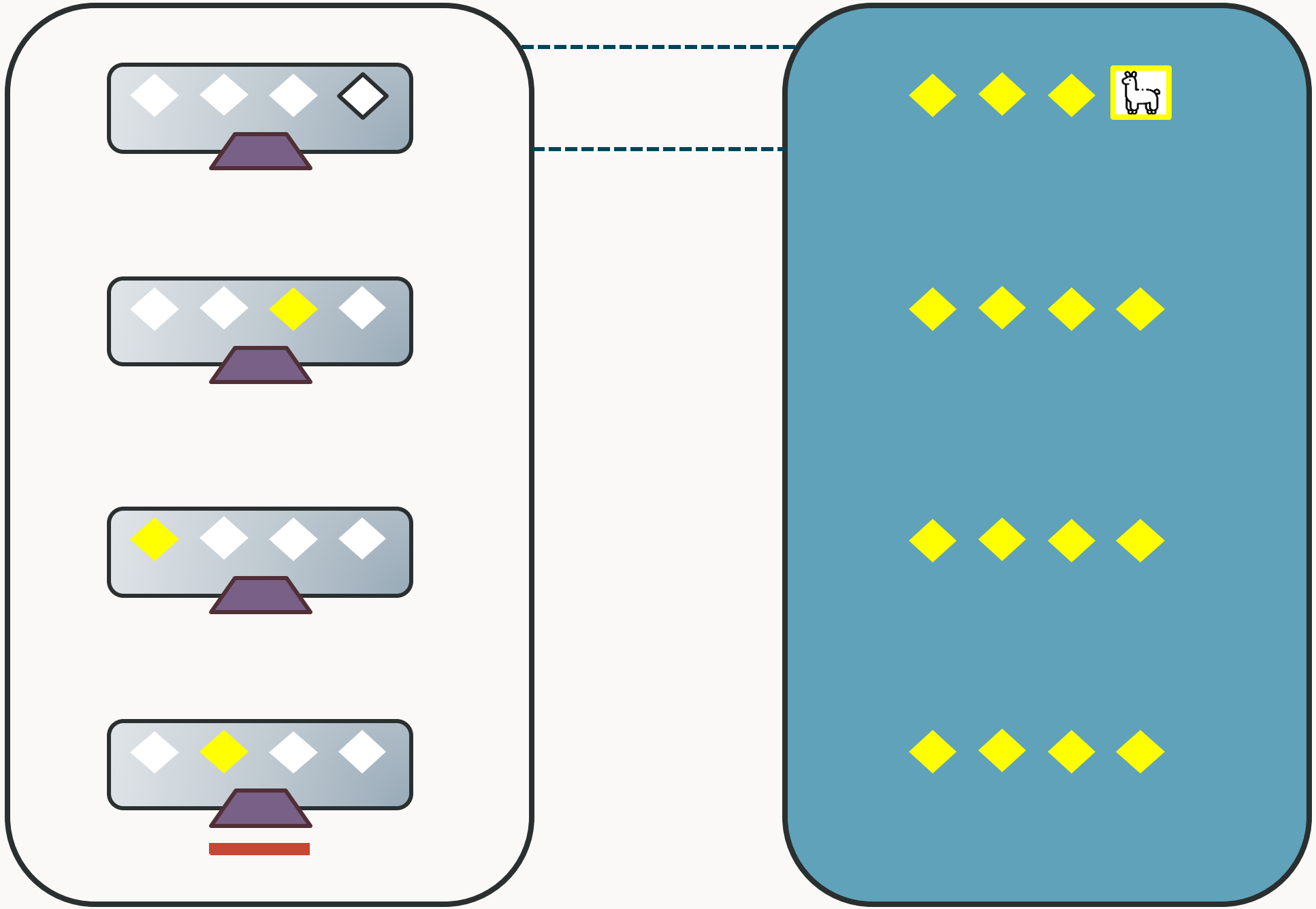


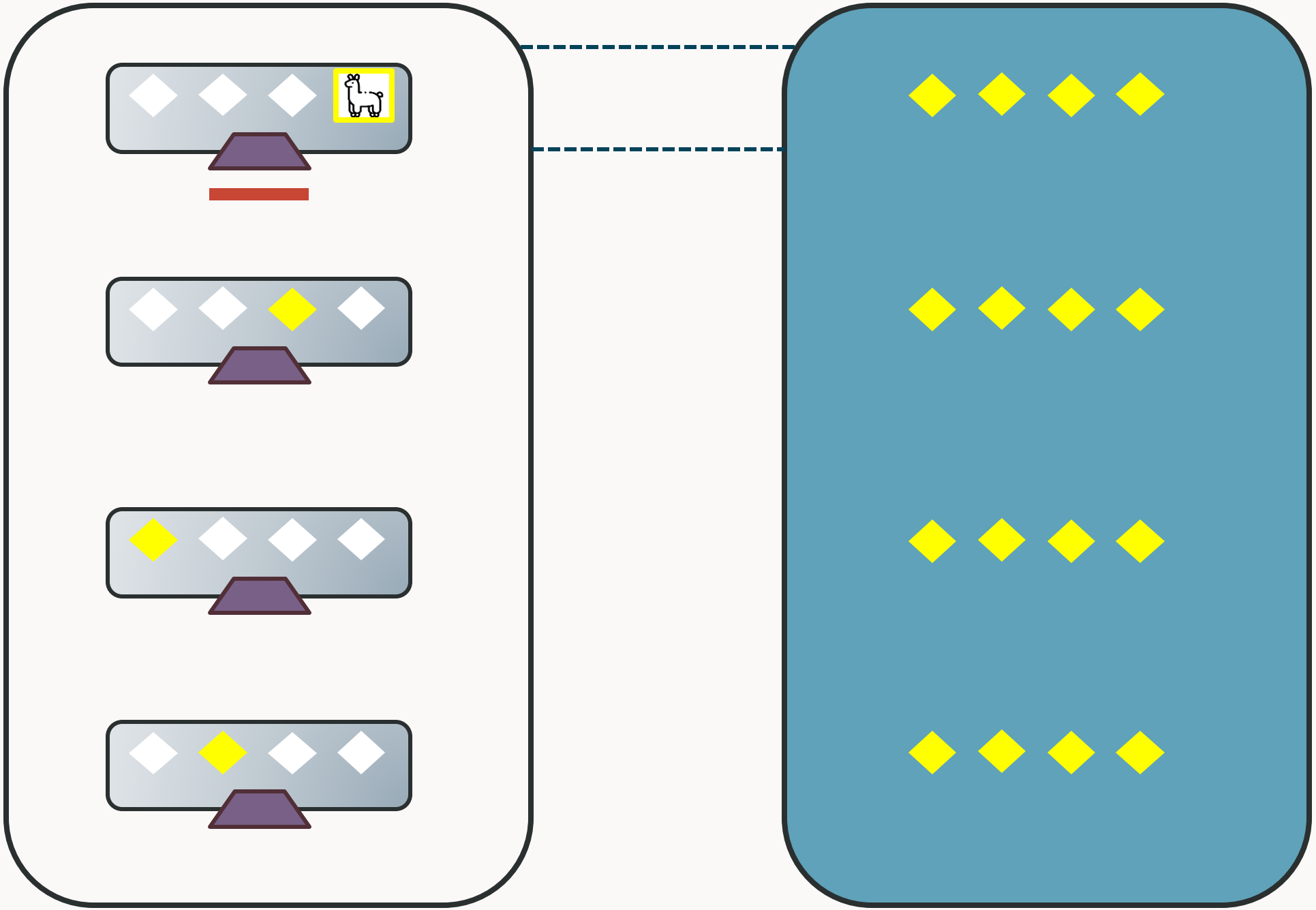


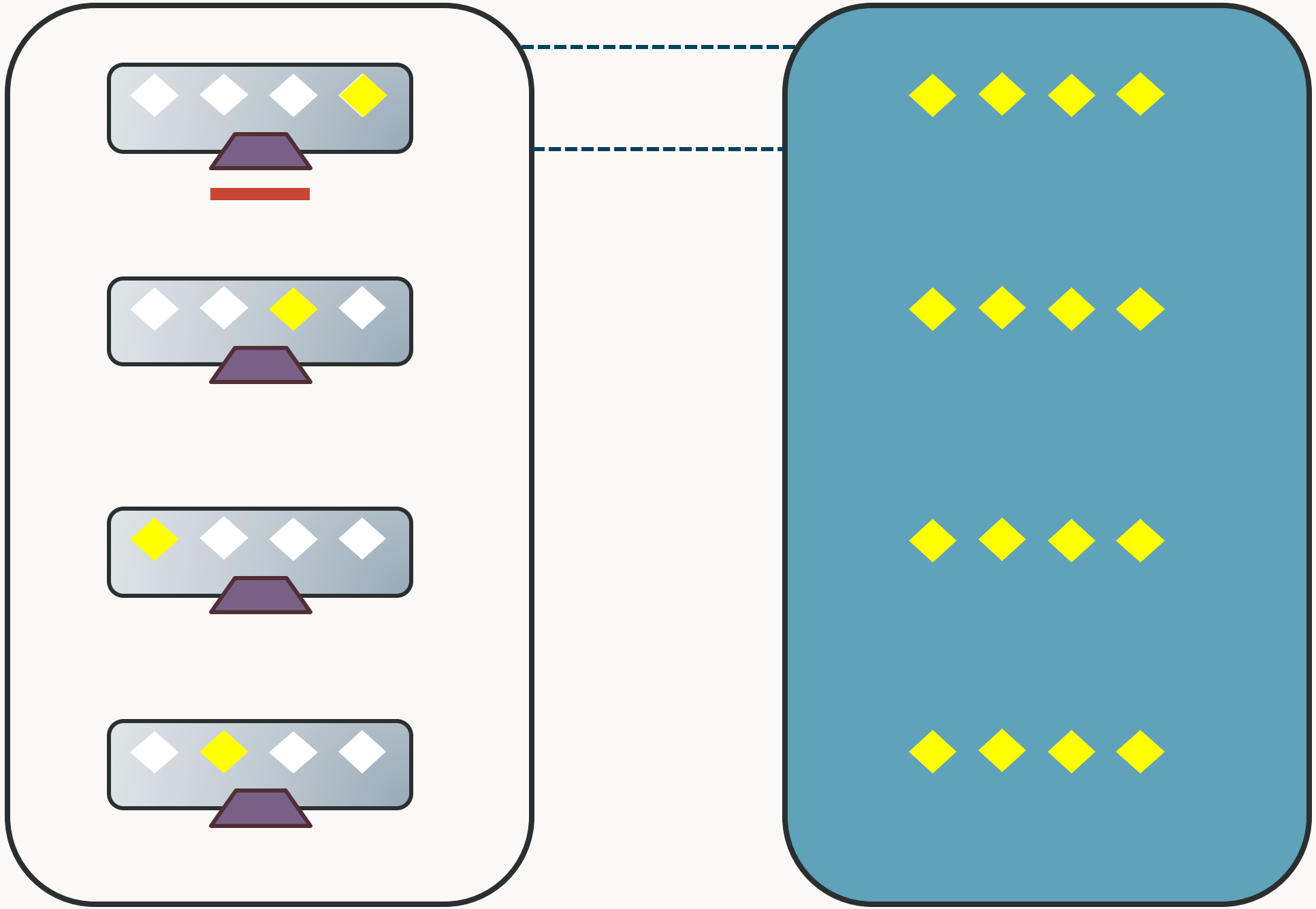




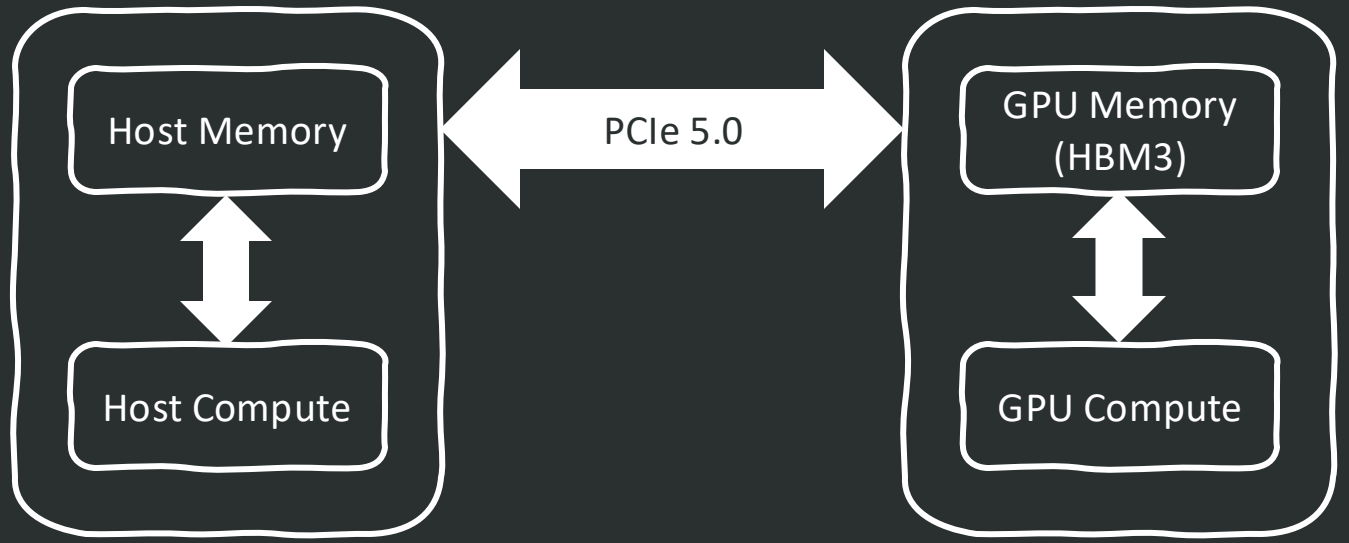




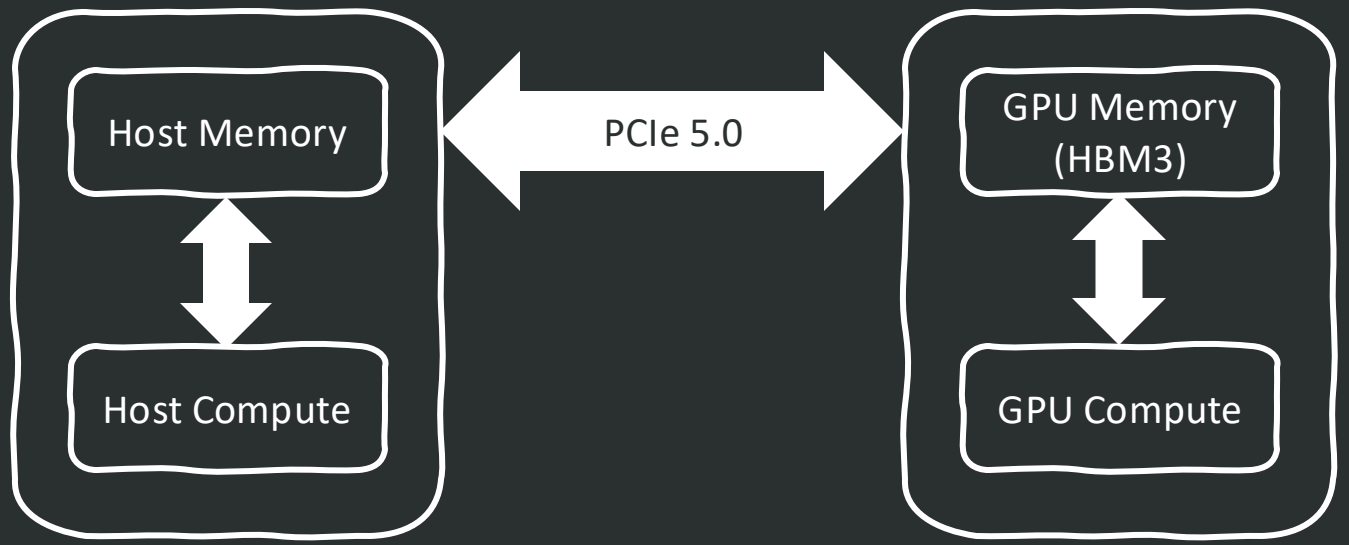




HtoD: Back-of-the-Envelope Calculations

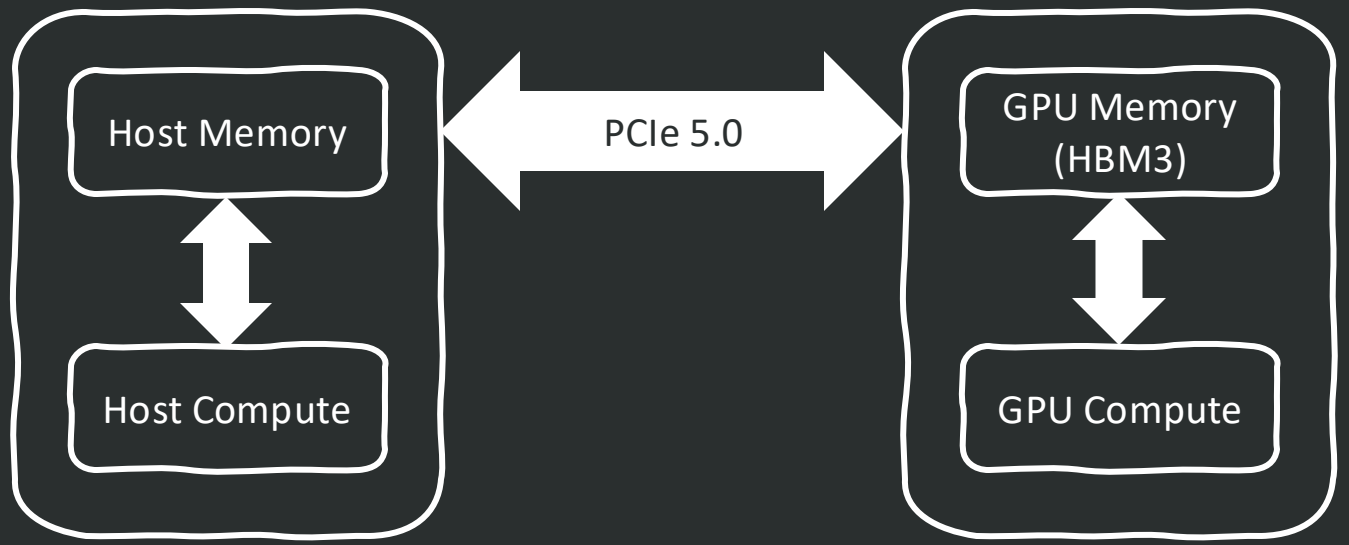


HtoD: Back-of-the-Envelope Calculations



- **64 GB/s** HtoD (and 64GB/s DtoH) transfer speed
- **0.251 GB/expert**

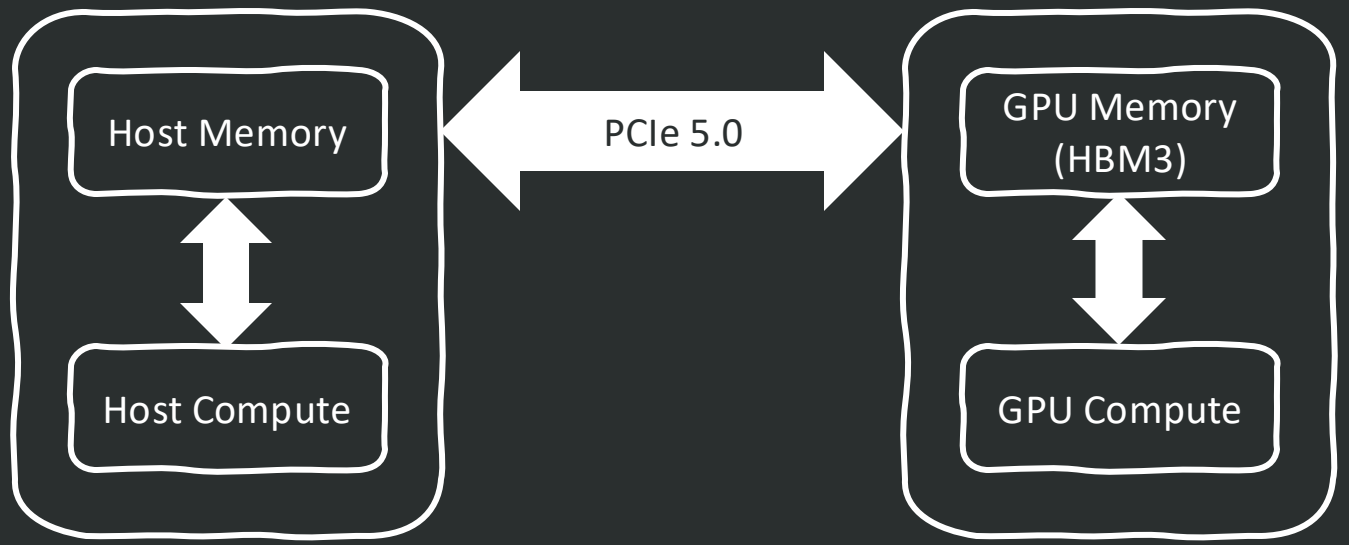
HtoD: Back-of-the-Envelope Calculations



- **64 GB/s** HtoD (and 64GB/s DtoH) transfer speed
- **0.251 GB/expert**

- Moving 1 expert HtoD theoretically takes:
  - (0.251658240 GB/expert) / (64 GB/s) = 0.00393216 s/expert = **3.932 ms/expert**

HtoD: Back-of-the-Envelope Calculations



- **64 GB/s** HtoD (and 64GB/s DtoH) transfer speed
- **0.251 GB/expert**

- Moving 1 expert HtoD theoretically takes:
  - (0.251658240 GB/expert) / (64 GB/s) = 0.00393216 s/expert = **3.932 ms/expert**

- ≅ 4x pure execution of Llama4DecoderLayer

Expert Prediction

Predicting Router Outputs at Future Layers

- **Expert Activation Paths** vs **Small Networks**

- Markov chain of expert predictions?
  - preliminary experiments showed that can't use router outputs (of dim 16) as features

- Predicting future hidden states (cf. self-speculative decoding)

- Several papers apply routers at layer $\ell + 1$ to activations at current layer ℓ to make a prefetching decision:
  - ***Pre-gated MoE****: An Algorithm-System Co-Design for Fast and Scalable Mixture-of-Expert Inference*
  - *Fast Inference of Mixture-of-Experts Language Models* ***with Offloading***
  - ***Fate****: Fast Edge Inference of Mixture-of-Experts Models via Cross-Layer Gate*

Finetuning Cloned Routers for δ Layers up (idea)

Finetuning Cloned Routers for δ Layers up (idea)

- Previous papers show (and we verify) that **a hidden state at layer ℓ can be fed into router at layer $\ell + 1$ and predict expert for layer $\ell + 1$ with decent accuracy**

Finetuning Cloned Routers for δ Layers up (idea)

- Previous papers show (and we verify) that **a hidden state at layer ℓ can be fed into router at layer $\ell + 1$ and predict expert for layer $\ell + 1$ with decent accuracy**

- Our extension:

Finetuning Cloned Routers for δ Layers up (idea)

- Previous papers show (and we verify) that **a hidden state at layer ℓ can be fed into router at layer $\ell + 1$ and predict expert for layer $\ell + 1$ with decent accuracy**

- Our extension:

  - **Generalize from router at layer $\ell + 1$ to router at layer $\ell + \delta$** for as large a δ as needed to allow for maximum copy-compute overlap while retaining predictive capacity

Finetuning Cloned Routers for δ Layers up (idea)

- Previous papers show (and we verify) that **a hidden state at layer ℓ can be fed into router at layer $\ell + 1$ and predict expert for layer $\ell + 1$ with decent accuracy**

- Our extension:

  - **Generalize from router at layer $\ell + 1$ to router at layer $\ell + \delta$** for as large a δ as needed to allow for maximum copy-compute overlap while retaining predictive capacity

  - **Clone the routers at layer $\ell + \delta$ and finetune them (*with bias, to account for layer skipping*)** on some held-out activations to better utilize activations at layer ℓ

Finetuning Cloned Routers for δ Layers up (idea)

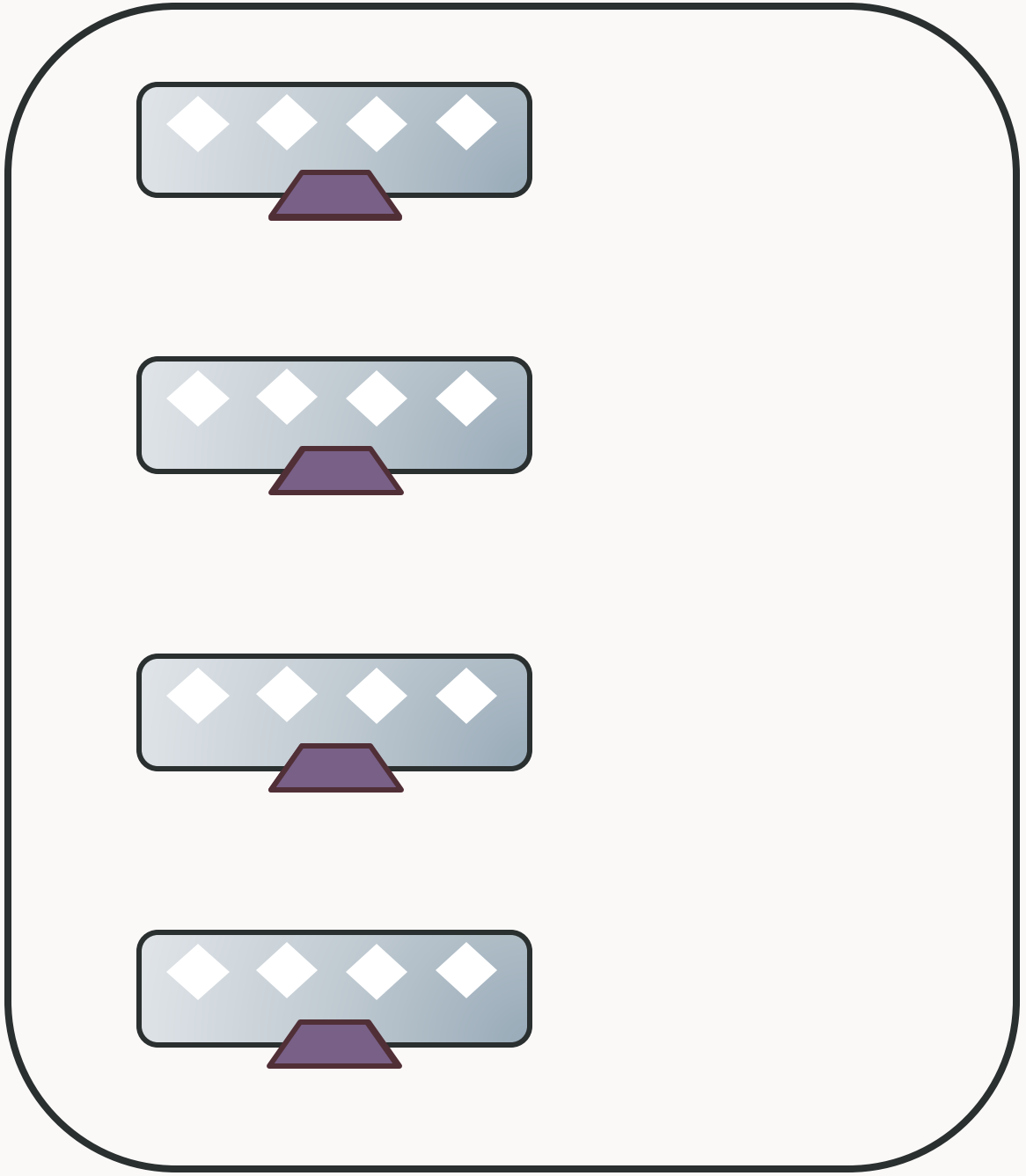
- Previous papers show (and we verify) that **a hidden state at layer ℓ can be fed into router at layer $\ell + 1$ and predict expert for layer $\ell + 1$ with decent accuracy**

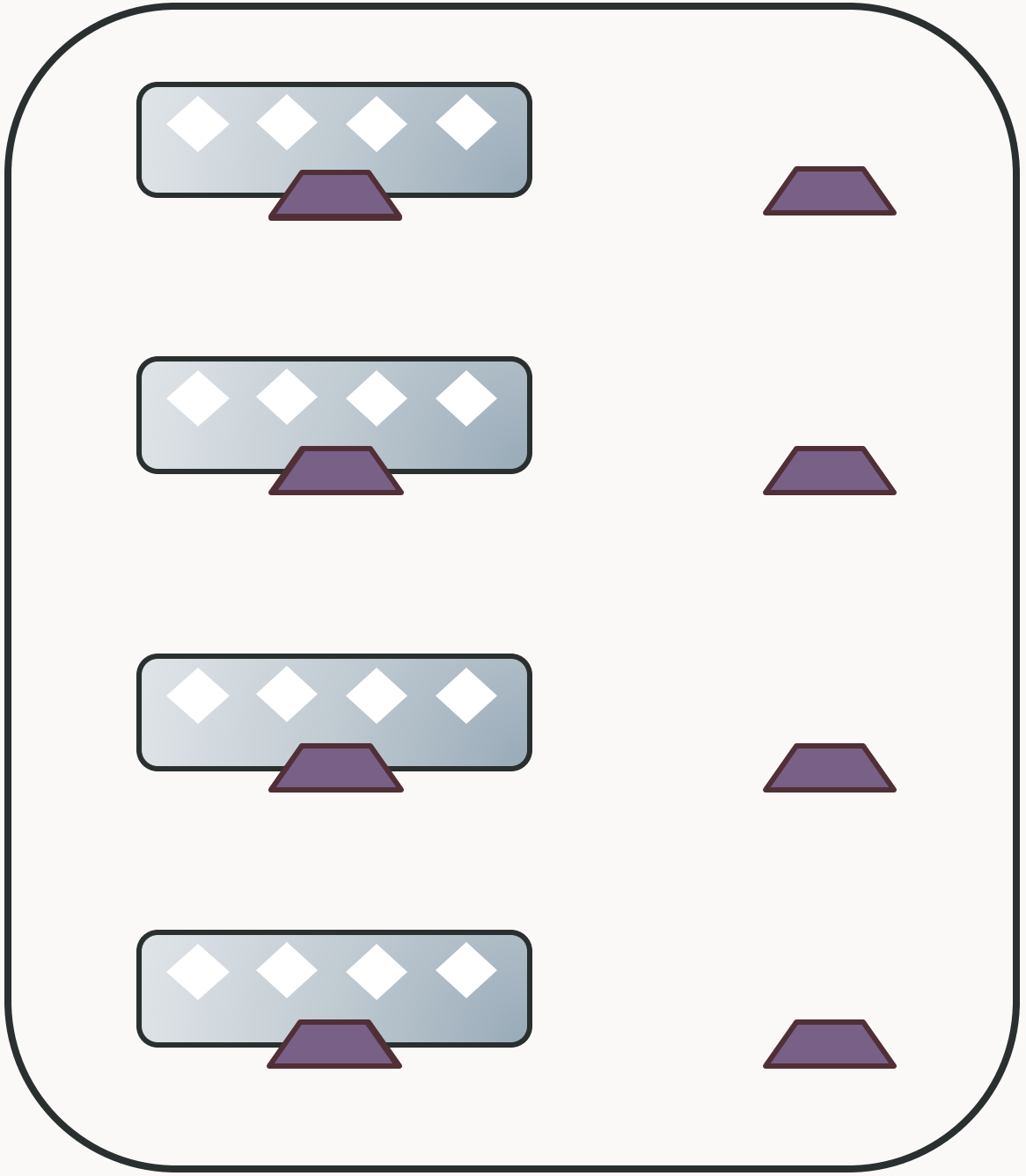
- Our extension:

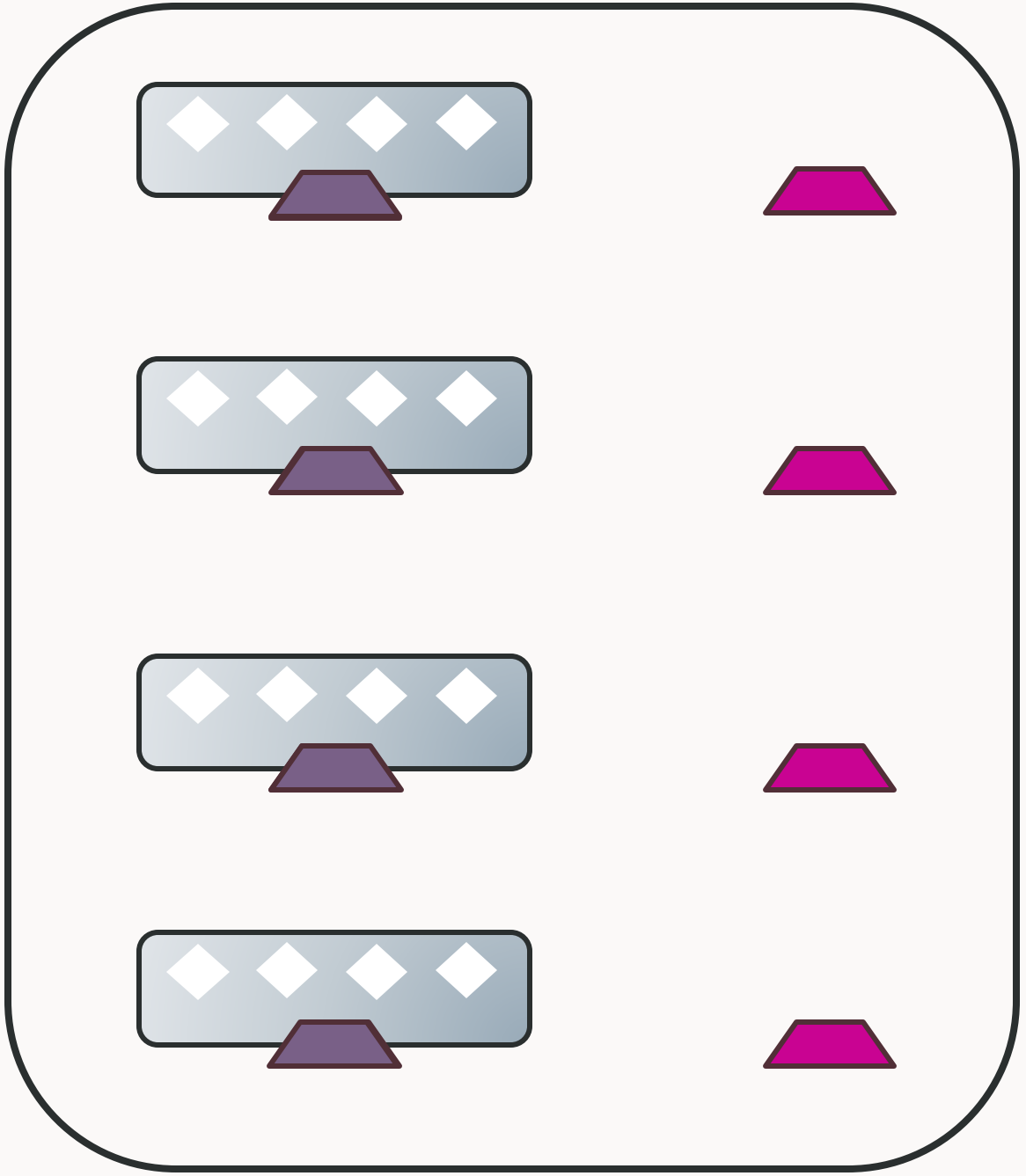
  - **Generalize from router at layer $\ell + 1$ to router at layer $\ell + \delta$** for as large a δ as needed to allow for maximum copy-compute overlap while retaining predictive capacity

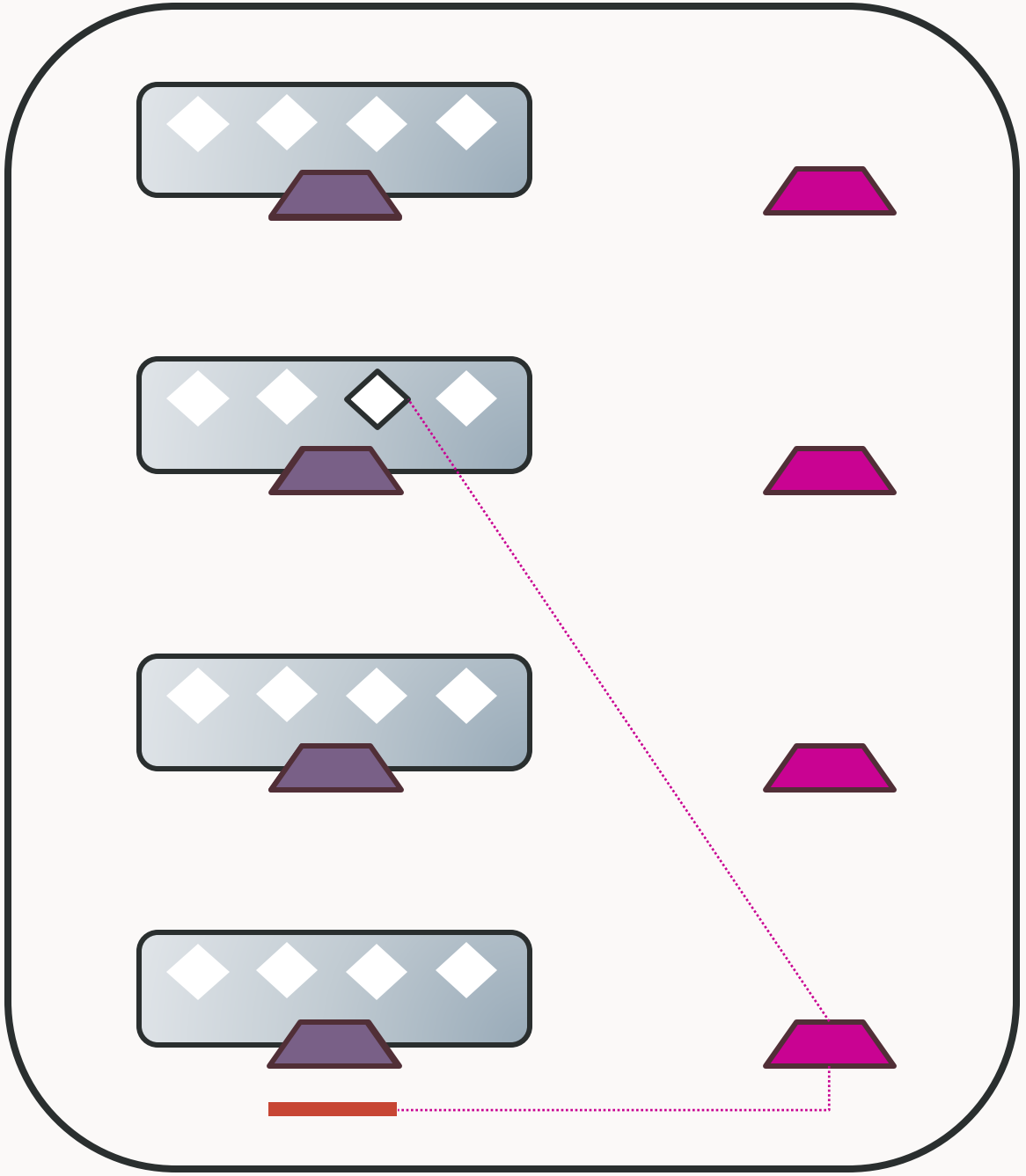
  - **Clone the routers at layer $\ell + \delta$ and finetune them (*with bias, to account for layer skipping*)** on some held-out activations to better utilize activations at layer ℓ

  - We are doubling the number of routers in our model, which amounts to an affordable extra ~8MB on GPU
    - $(5120 \times 16)_{\text{params/router}} * 2_{\text{bytes/bfloat16}} * 48_{\text{routers/model}} = 7.864320$ MB









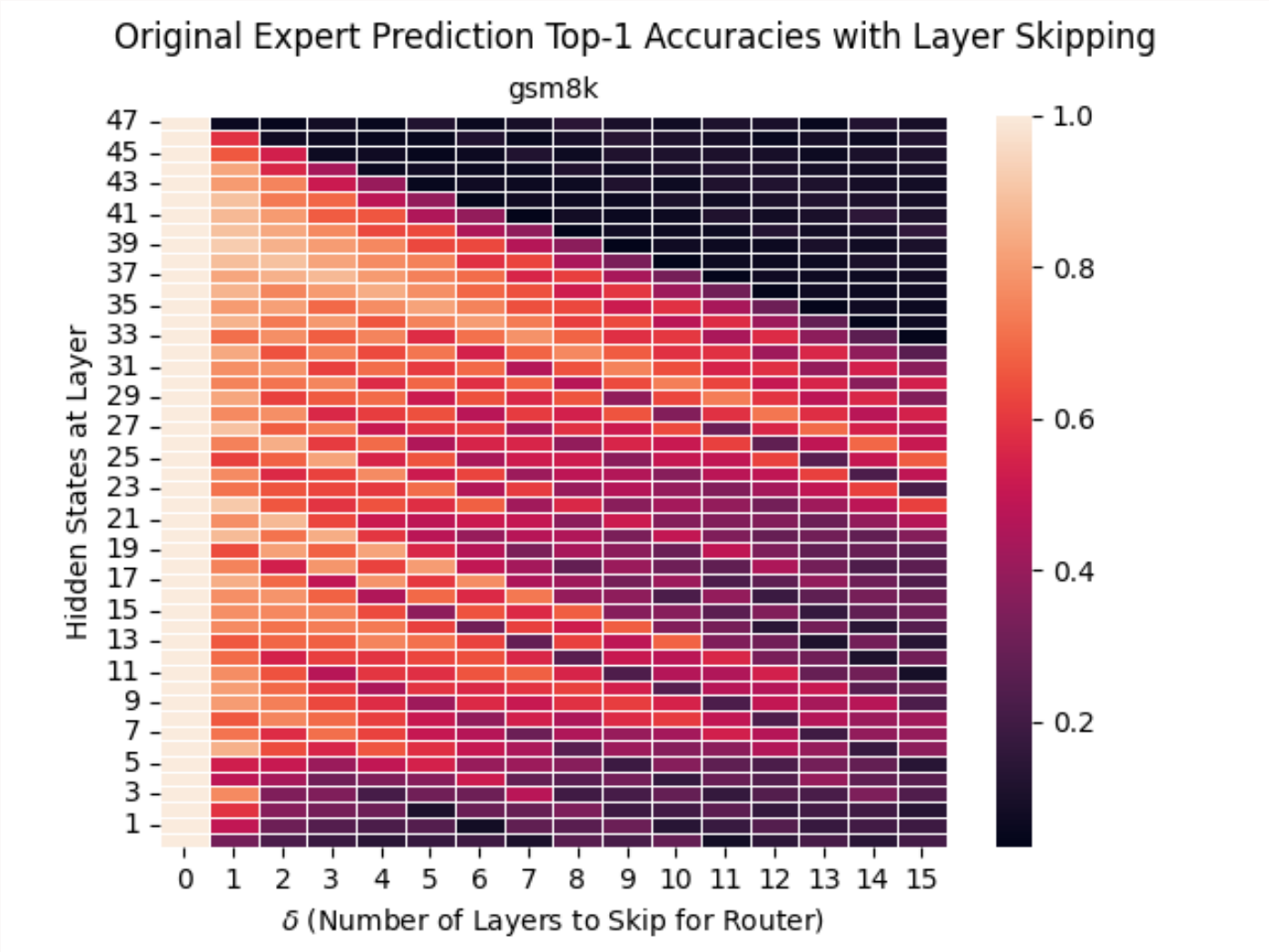
Finetuning Cloned Routers for δ Layers up (procedure)

- Collect router input-output activations for ~115K tokens for both “awesome-chatgpt-prompts” and “gsm8k”
- Each cloned router gets finetuned inside its own Optuna study (100 trials):

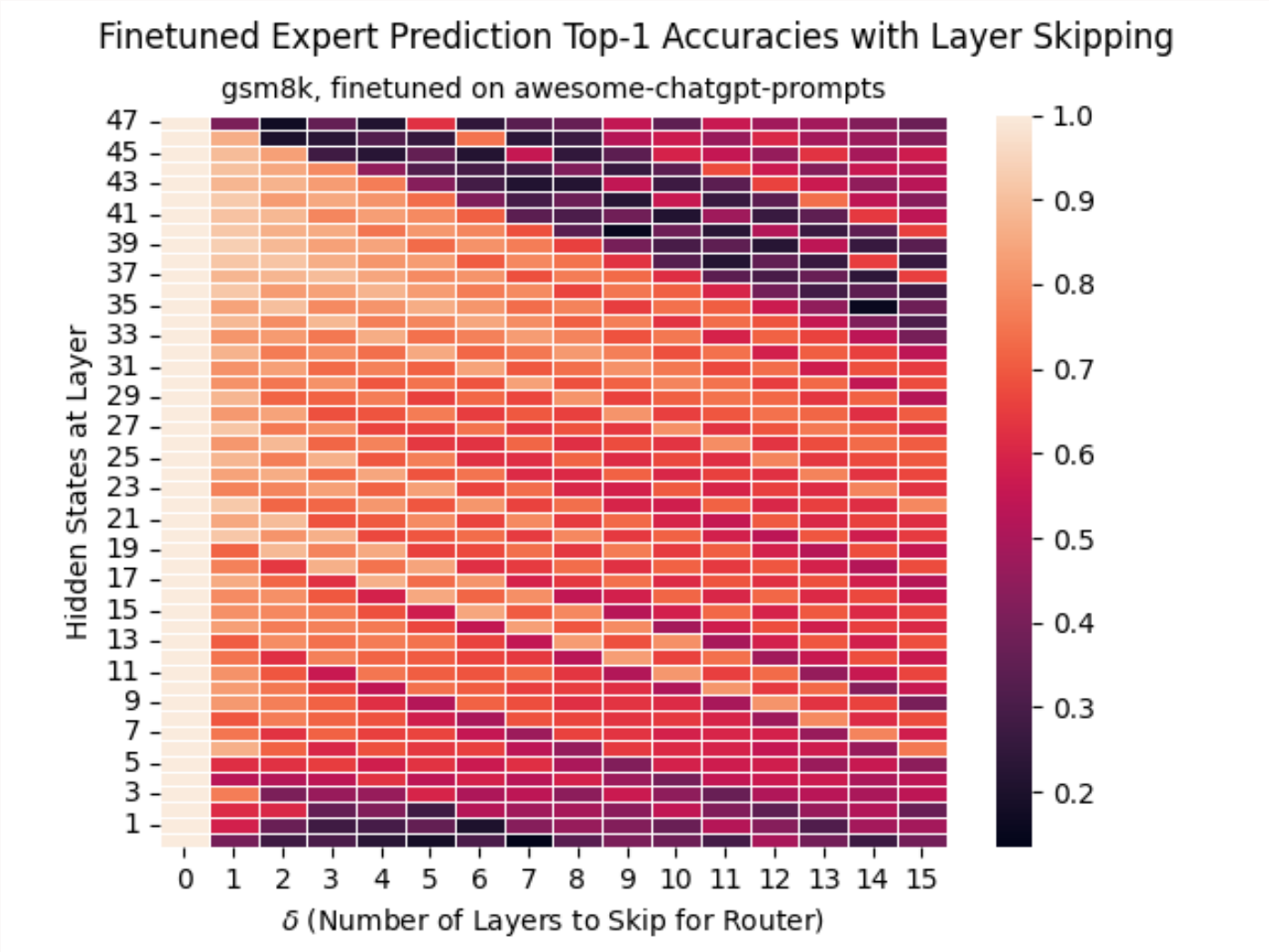
```
lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)
optimizer_name = trial.suggest_categorical("optimizer", ["Adam", "RMSprop"])
loss_fn_name = trial.suggest_categorical("loss_fn", ["CrossEntropyLoss", "MSELoss"])
```

- At every epoch, evaluate top-1 accuracy on “opposite” dataset:
  - if training on “awesome-chatgpt-prompts” activations, eval on “gsm8k” and vice versa
- 25K epochs with early stopping if best accuracy doesn’t improve within 50 epochs
- Best hyperparameter combination gets used to train and serialize best version of router for layer/delta
- Serialized routers get loaded during inference to predict which experts to prefetch

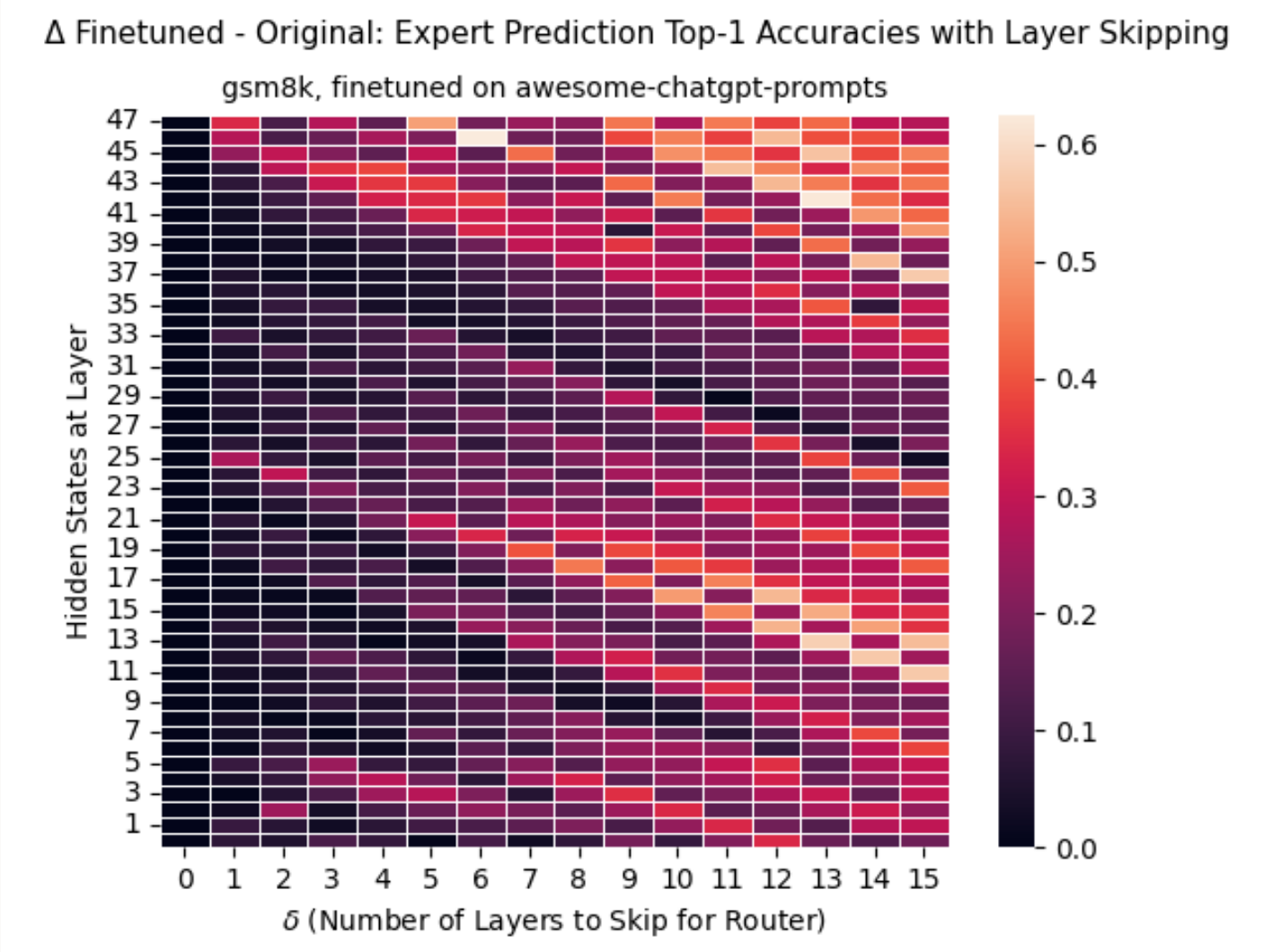
Llama4Scout: Expert Prediction with Layer Skipping (Original Routers)



Llama4Scout: Expert Prediction with Layer Skipping (Finetuned Routers)



Llama4Scout: Expert Prediction with Layer Skipping (Improvement)



Baselines

Adapting Current Implementation to 1 H100

(vLLM implementation can’t run on a single H100 unmodified)

Adapting Current Implementation to 1 H100

(vLLM implementation can't run on a single H100 unmodified)

- By default, all expert weights get created on GPU — causes CUDA OOM if trying to run Llama 4 Scout on 1 H100

```
        # Fused gate_up_proj (column parallel)
        w13_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            2 * intermediate_size_per_partition,
            hidden_size,
            dtype=params_dtype),
                                        requires_grad=False)
```

```
        # down_proj (row parallel)
        w2_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            hidden_size,
            intermediate_size_per_partition,
            dtype=params_dtype),
                                        requires_grad=False)
```

Adapting Current Implementation to 1 H100

(vLLM implementation can't run on a single H100 unmodified)

- By default, all expert weights get created on GPU — causes CUDA OOM if trying to run Llama 4 Scout on 1 H100

```
        # Fused gate_up_proj (column parallel)
        w13_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            2 * intermediate_size_per_partition,
            hidden_size,
            dtype=params_dtype),
                                        requires_grad=False)
```

```
        # down_proj (row parallel)
        w2_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            hidden_size,
            intermediate_size_per_partition,
            dtype=params_dtype),
                                        requires_grad=False)
```

- To circumvent this, we must set expert weights to be created and stored on CPU

```
        # Fused gate_up_proj (column parallel)
        w13_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            2 * intermediate_size_per_partition,
            hidden_size,
            dtype=params_dtype,
            device="cpu",
            pin_memory=True),
                                        requires_grad=False)
```

```
        # down_proj (row parallel)
        w2_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            hidden_size,
            intermediate_size_per_partition,
            dtype=params_dtype,
            device="cpu",
            pin_memory=True),
                                        requires_grad=False)
```

Adapting Current Implementation to 1 H100

(vLLM implementation can't run on a single H100 unmodified)

- By default, all expert weights get created on GPU — causes CUDA OOM if trying to run Llama 4 Scout on 1 H100

```
        # Fused gate_up_proj (column parallel)
        w13_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            2 * intermediate_size_per_partition,
            hidden_size,
            dtype=params_dtype),
                                        requires_grad=False)
```

```
        # down_proj (row parallel)
        w2_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            hidden_size,
            intermediate_size_per_partition,
            dtype=params_dtype),
                                        requires_grad=False)
```

- To circumvent this, we must set expert weights to be created and stored on CPU

```
        # Fused gate_up_proj (column parallel)
        w13_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            2 * intermediate_size_per_partition,
            hidden_size,
            dtype=params_dtype,
            device="cpu",
            pin_memory=True),
                                        requires_grad=False)
```

```
        # down_proj (row parallel)
        w2_weight = torch.nn.Parameter(torch.empty(
            num_experts,
            hidden_size,
            intermediate_size_per_partition,
            dtype=params_dtype,
            device="cpu",
            pin_memory=True),
                                        requires_grad=False)
```

- The triton `fused_moe_kernel` loads predicted experts from CPU on-demand for every layer for every token:

```
off_experts = tl.load(expert_ids_ptr + pid_m).to(tl.int64)
```

“Ideal” Scenario: No CPU-to-GPU (HtoD) Data Movement at All

(Compute-Bound)

“Ideal” Scenario: No CPU-to-GPU (HtoD) Data Movement at All

(Compute-Bound)

Assume, unrealistically…

i. we could fit all experts onto a single GPU

OR

ii. each layer used a single fixed expert (residing on GPU), requiring no memory movement

“Ideal” Scenario: No CPU-to-GPU (HtoD) Data Movement at All

(Compute-Bound)

Assume, unrealistically…

i. we could fit all experts onto a single GPU

OR

ii. each layer used a single fixed expert (residing on GPU), requiring no memory movement

Then…

Profiling inference while using a single dummy expert to simulate no HtoD movement…

- **Llama4DecoderLayer** execution takes 1218500.8360339506 ns ≅ **1.22 ms**
- (x 48) → **Llama4ForConditionalGeneration** takes 58870344.46296296 ns ≅ **58.87 ms**

“Ideal” Scenario: No CPU-to-GPU (HtoD) Data Movement at All

(Compute-Bound)

Assume, unrealistically…

i. we could fit all experts onto a single GPU

OR

ii. each layer used a single fixed expert (residing on GPU), requiring no memory movement

Then…

Profiling inference while using a single dummy expert to simulate no HtoD movement…

- **Llama4DecoderLayer** execution takes 1218500.8360339506 ns ≅ **1.22 ms**
- (x 48) → **Llama4ForConditionalGeneration** takes 58870344.46296296 ns ≅ **58.87 ms**

We won’t ever go faster than computation without data transfer — **ABSOLUTE LOWER BOUND**

“Best-Case” Scenario: Expert Prefetching

(Memory-Bound)

“Best-Case” Scenario: Expert Prefetching

(Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the oracle-expected expert for initial token(s)

“Best-Case” Scenario: Oracle Expert Prefetching

(Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the oracle-expected expert for initial token(s)

- For subsequent tokens:

“Best-Case” Scenario: Oracle Expert Prefetching

(Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the oracle-expected expert for initial token(s)

- For subsequent tokens:

  - if oracle-expected expert at layer $\ell + \delta$ *is not* already in cache, then **asynchronously prefetch expert δ layers in advance** in non-default CUDA Stream

“Best-Case” Scenario: Oracle Expert Prefetching

(Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the oracle-expected expert for initial token(s)
- For subsequent tokens:
  - if oracle-expected expert at layer $\ell + \delta$ *is not* already in cache, then **asynchronously prefetch expert δ layers in advance** in non-default CUDA Stream
  - if expert expected at layer $\ell + \delta$ *is* already in cache, then **do nothing**

“Best-Case” Scenario: Oracle Expert Prefetching

(Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the oracle-expected expert for initial token(s)
- For subsequent tokens:
  - if oracle-expected expert at layer $\ell + \delta$ *is not* already in cache, then **asynchronously prefetch expert δ layers in advance** in non-default CUDA Stream
  - if expert expected at layer $\ell + \delta$ *is* already in cache, then **do nothing**
  - at layer ℓ, **synchronize** to ensure asynchronous copy triggered at $\ell - \delta$ has completed

“Best-Case” Scenario: Oracle Expert Prefetching

(Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the oracle-expected expert for initial token(s)
- For subsequent tokens:
  - if oracle-expected expert at layer $\ell + \delta$ *is not* already in cache, then **asynchronously prefetch expert δ layers in advance** in non-default CUDA Stream
  - if expert expected at layer $\ell + \delta$ *is* already in cache, then **do nothing**
  - at layer ℓ, **synchronize** to ensure asynchronous copy triggered at $\ell - \delta$ has completed
- For **last δ tokens of generation, don’t do any prefetching**

“Best-Case” Scenario: Oracle Expert Prefetching

(Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the oracle-expected expert for initial token(s)
- For subsequent tokens:
  - if oracle-expected expert at layer $\ell + \delta$ *is not* already in cache, then **asynchronously prefetch expert δ layers in advance** in non-default CUDA Stream
  - if expert expected at layer $\ell + \delta$ *is* already in cache, then **do nothing**
  - at layer ℓ, **synchronize** to ensure asynchronous copy triggered at $\ell - \delta$ has completed
- For **last δ tokens of generation, don’t do any prefetching**

This is the best we can do given dynamic MoE expert decisions — **ACTUAL LOWER BOUND**

“Realistic” Scenario: ML-Based Expert Prefetching

(Predictability- & Memory-Bound)

“Realistic” Scenario: ML-Based Expert Prefetching

(Predictability- & Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the **most likely** expert for initial token(s)

“Realistic” Scenario: ML-Based Expert Prefetching

(Predictability- & Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the **most likely** expert for initial token(s)

- For subsequent tokens:

“Realistic” Scenario: ML-Based Expert Prefetching

(Predictability- & Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the **most likely** expert for initial token(s)
- For subsequent tokens:
  - at each layer ℓ, **use ML component** to predict which expert will be required δ layers ahead

“Realistic” Scenario: ML-Based Expert Prefetching

(Predictability- & Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the **most likely** expert for initial token(s)
- For subsequent tokens:
  - at each layer ℓ, **use ML component** to predict which expert will be required δ layers ahead
  - if expert expected at layer $\ell + \delta$ *is not* already in cache, then **asynchronously prefetch expert δ layers in advance** in non-default CUDA Stream

“Realistic” Scenario: ML-Based Expert Prefetching

(Predictability- & Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the **most likely** expert for initial token(s)
- For subsequent tokens:
  - at each layer ℓ, **use ML component** to predict which expert will be required δ layers ahead
  - if expert expected at layer $\ell + \delta$ *is not* already in cache, then **asynchronously prefetch expert δ layers in advance** in non-default CUDA Stream
  - if expert expected at layer $\ell + \delta$ *is* already in cache, then **do nothing**

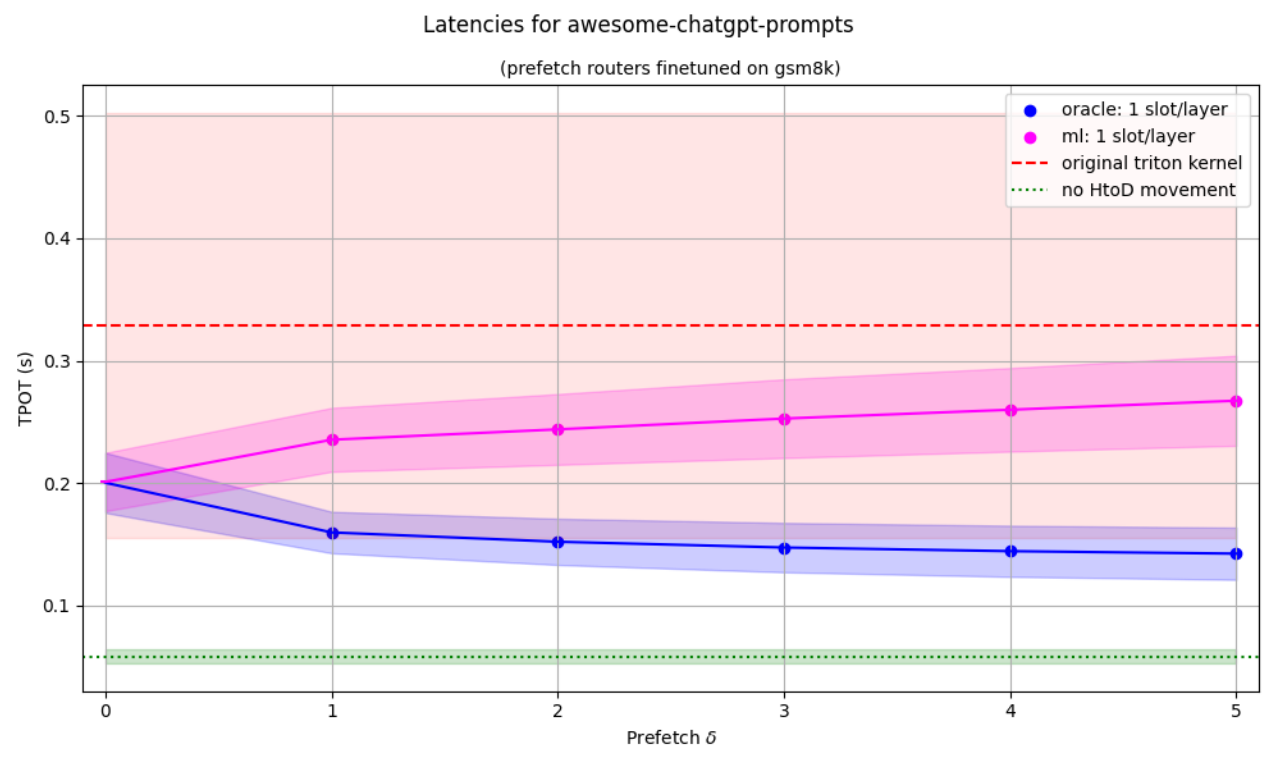
“Realistic” Scenario: ML-Based Expert Prefetching

(Predictability- & Memory-Bound)

- Before TG, **pre-populate** each layer’s cache with the **most likely** expert for initial token(s)
- For subsequent tokens:
  - at each layer ℓ, **use ML component** to predict which expert will be required δ layers ahead
  - if expert expected at layer $\ell + \delta$ *is not* already in cache, then **asynchronously prefetch expert δ layers in advance** in non-default CUDA Stream
  - if expert expected at layer $\ell + \delta$ *is* already in cache, then **do nothing**
  - at layer ℓ, **synchronize** to ensure asynchronous copy triggered at $\ell - \delta$ has completed

Latency Measurements

One Cache Slot per Layer



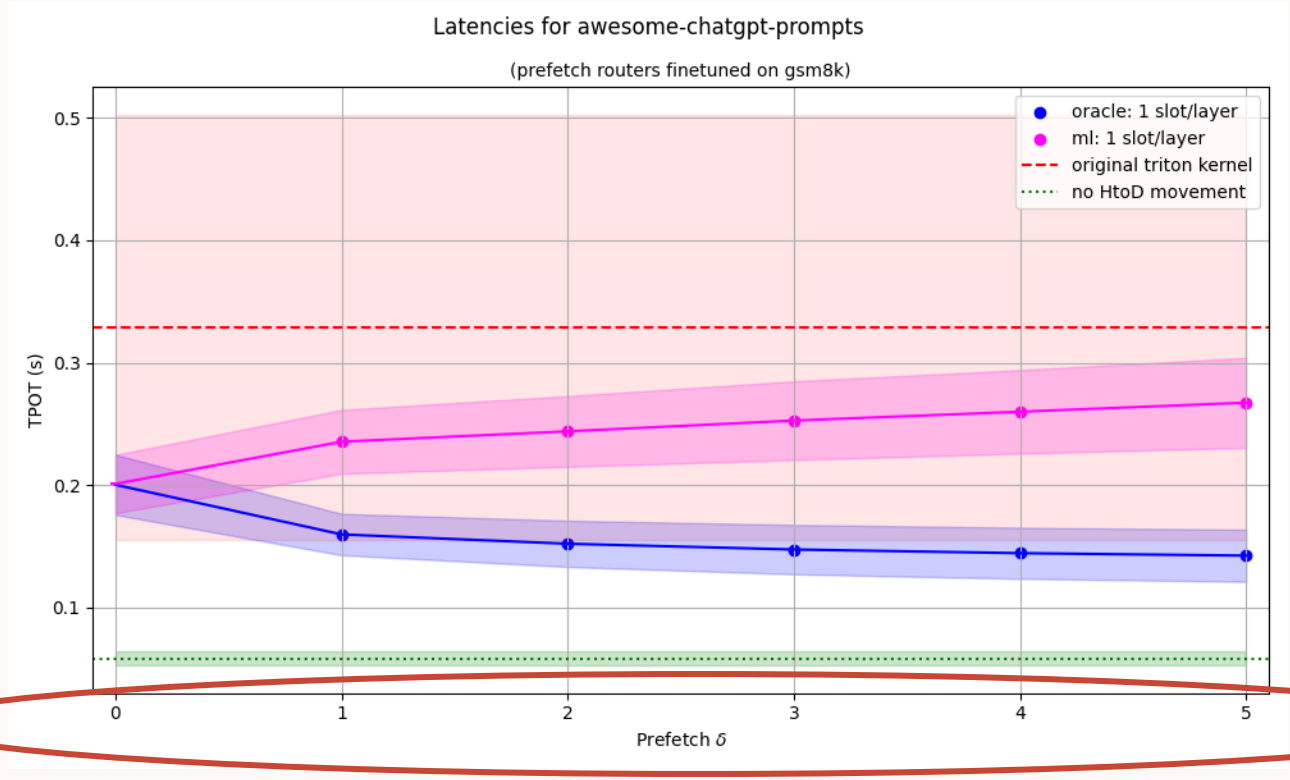
Cache miss rate

0.59	0.26	0.34	0.39	0.42	0.46
0.59					

Prefetch rate

0.63	0.65	0.67	0.68	0.7
0.59	0.59	0.59	0.59	0.59

One Cache Slot per Layer



i.e. if at layer ℓ, prefetch for δ layers up

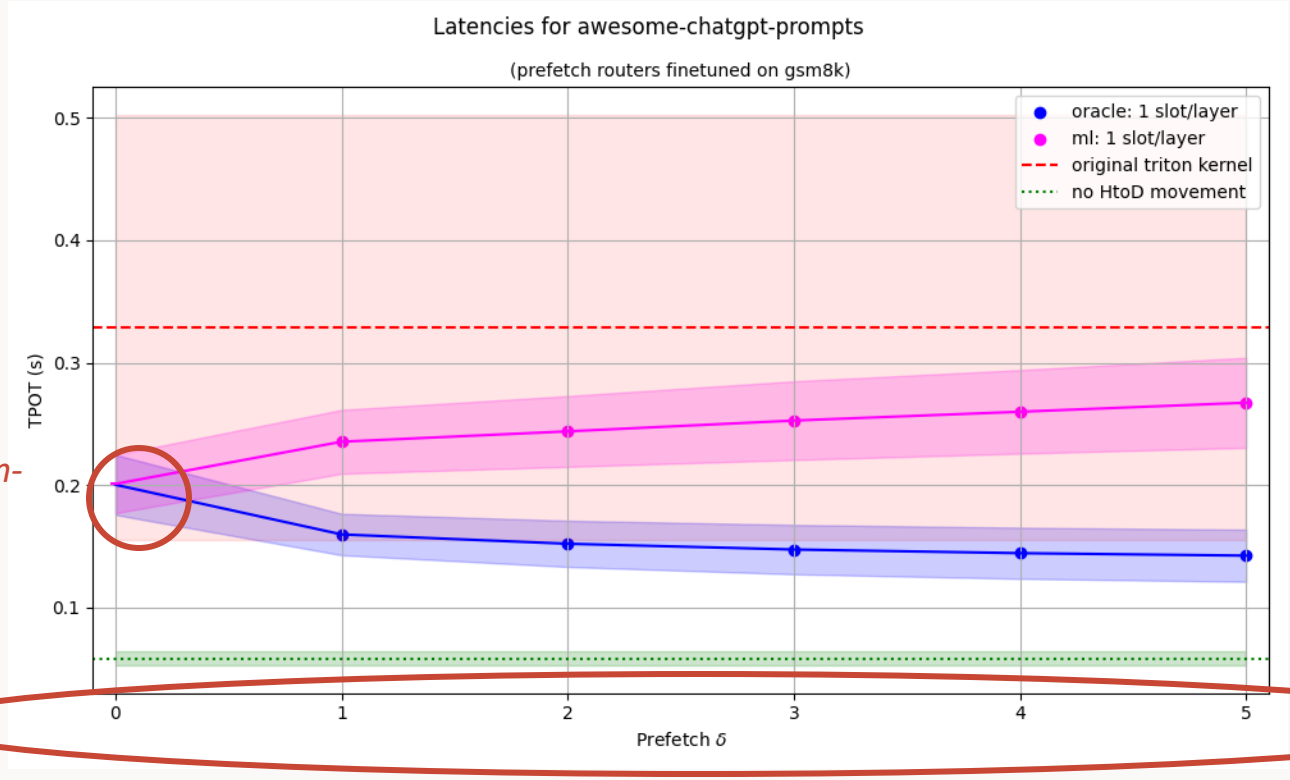
Cache miss rate

0.59 0.26 0.34 0.39 0.42 0.46
0.59

Prefetch rate

0.63 0.65 0.67 0.68 0.7
0.59 0.59 0.59 0.59 0.59

One Cache Slot per Layer



δ = 0 means no prefetching, only on-demand loading if cache miss

i.e. if at layer ℓ, prefetch for δ layers up

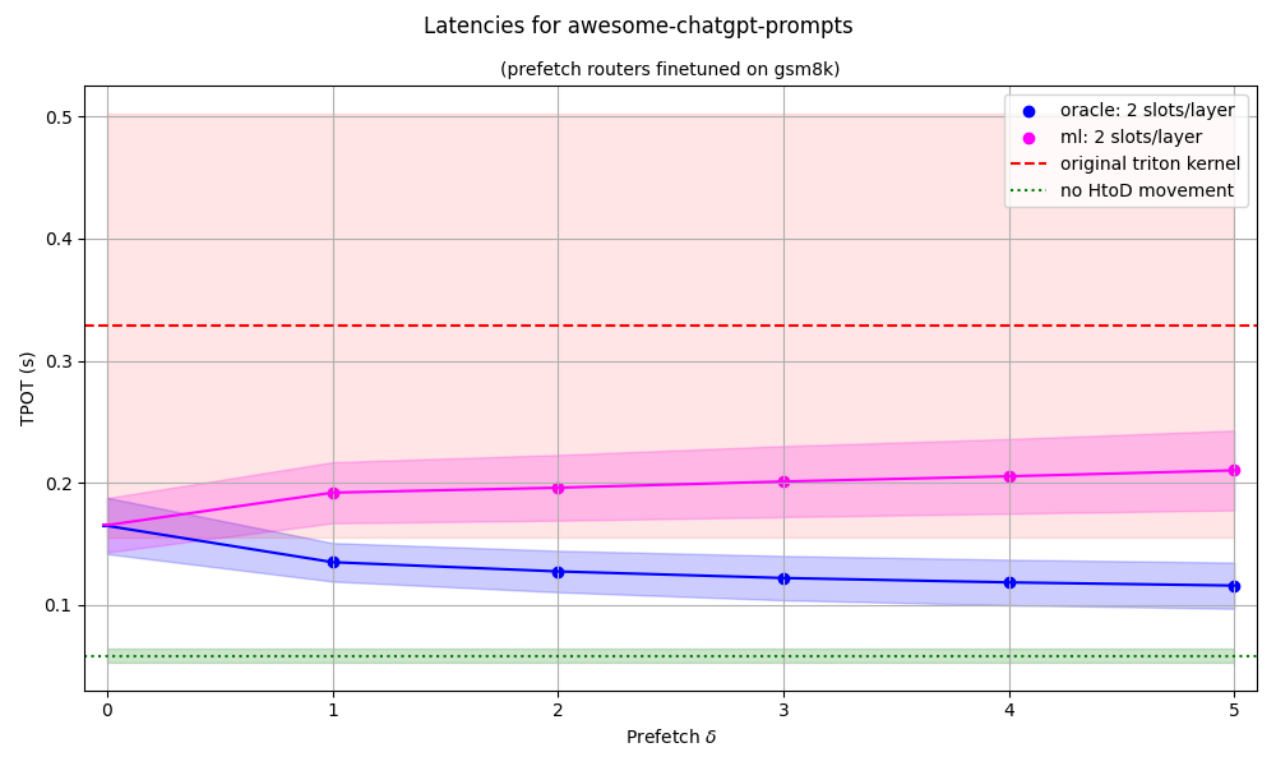
Cache miss rate

0.59 0.26 0.34 0.39 0.42 0.46
0.59

Prefetch rate

0.63 0.65 0.67 0.68 0.7
0.59 0.59 0.59 0.59 0.59

Two Cache Slots per Layer



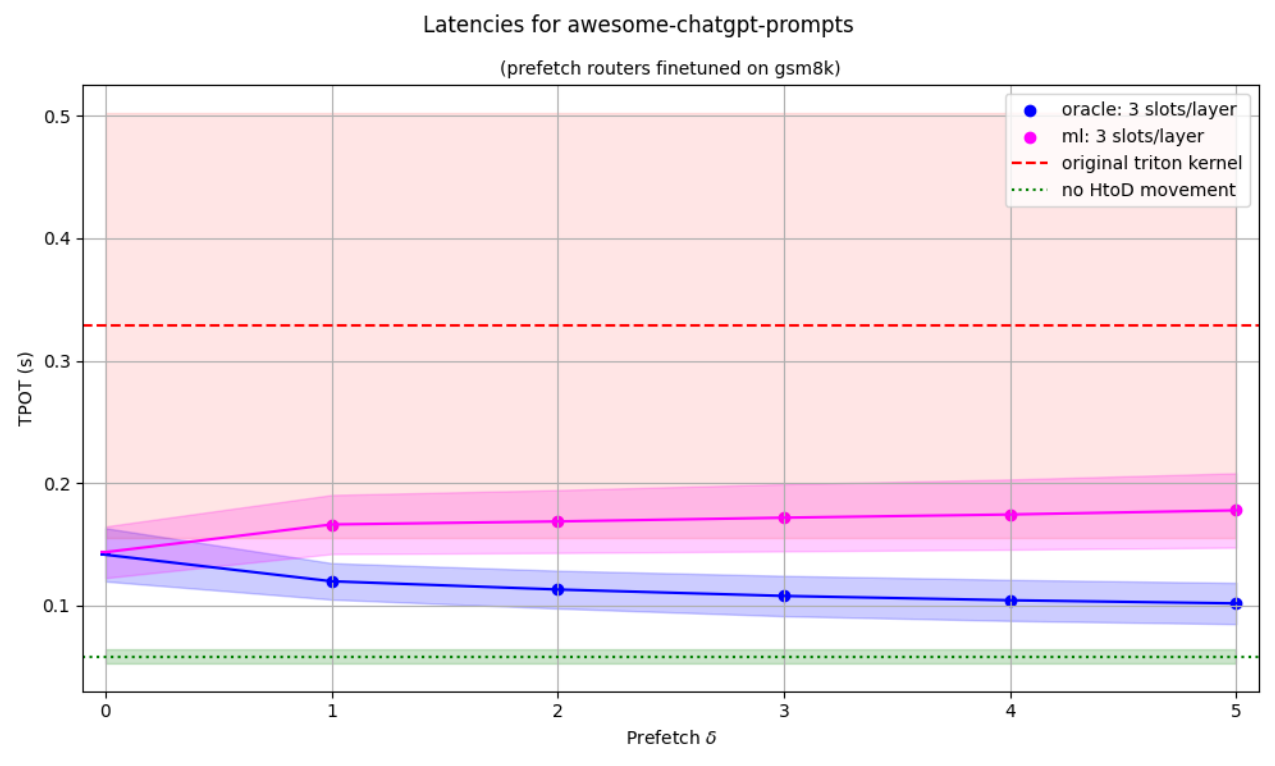
Cache miss rate

0.44	0.19	0.24	0.28	0.3	0.33
0.44					

Prefetch rate

0.48	0.5	0.53	0.53	0.55
0.43	0.43	0.43	0.43	0.43

Three Cache Slots per Layer



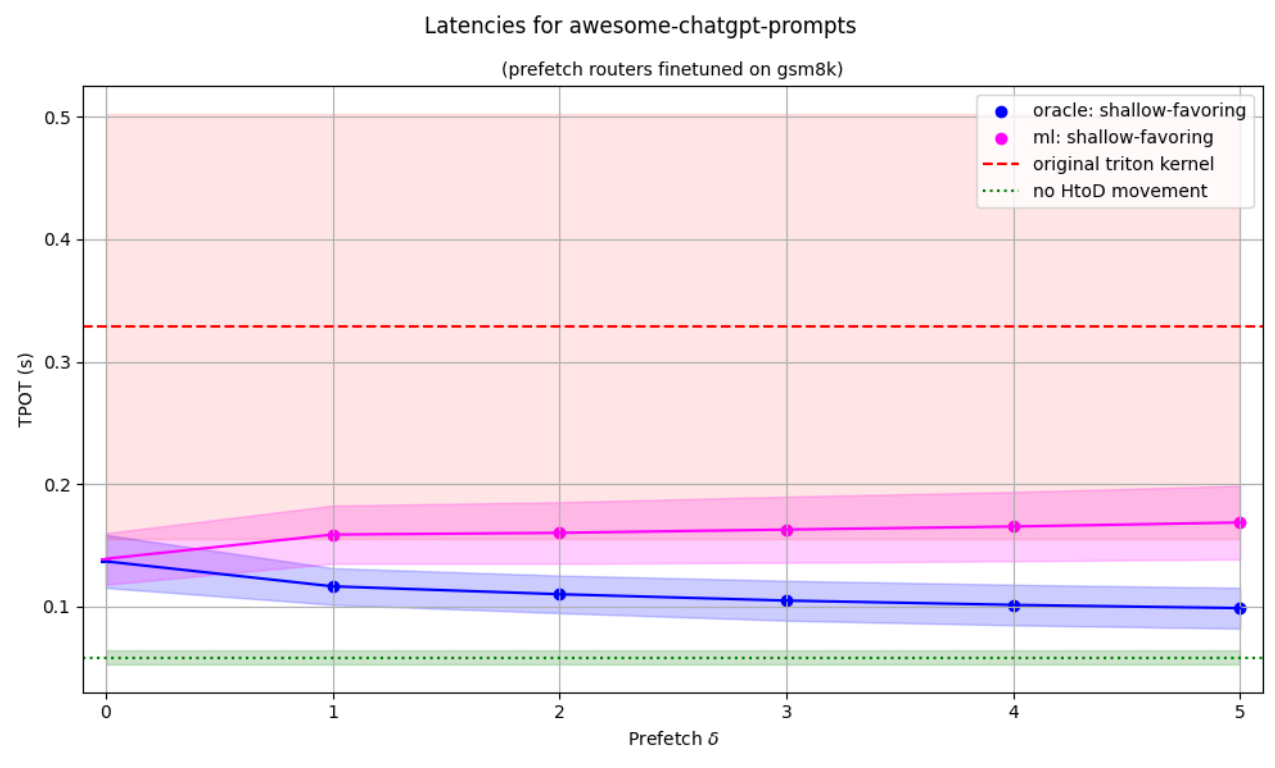
Cache miss rate

0.34	0.15	0.19	0.22	0.24	0.26
0.34					

Prefetch rate

0.38	0.41	0.43	0.43	0.45
0.33	0.33	0.33	0.33	0.33

“Shallow-Favoring”: 5 Cache Slots at Layers 0-5, 3 cache slots at Layers 6-47



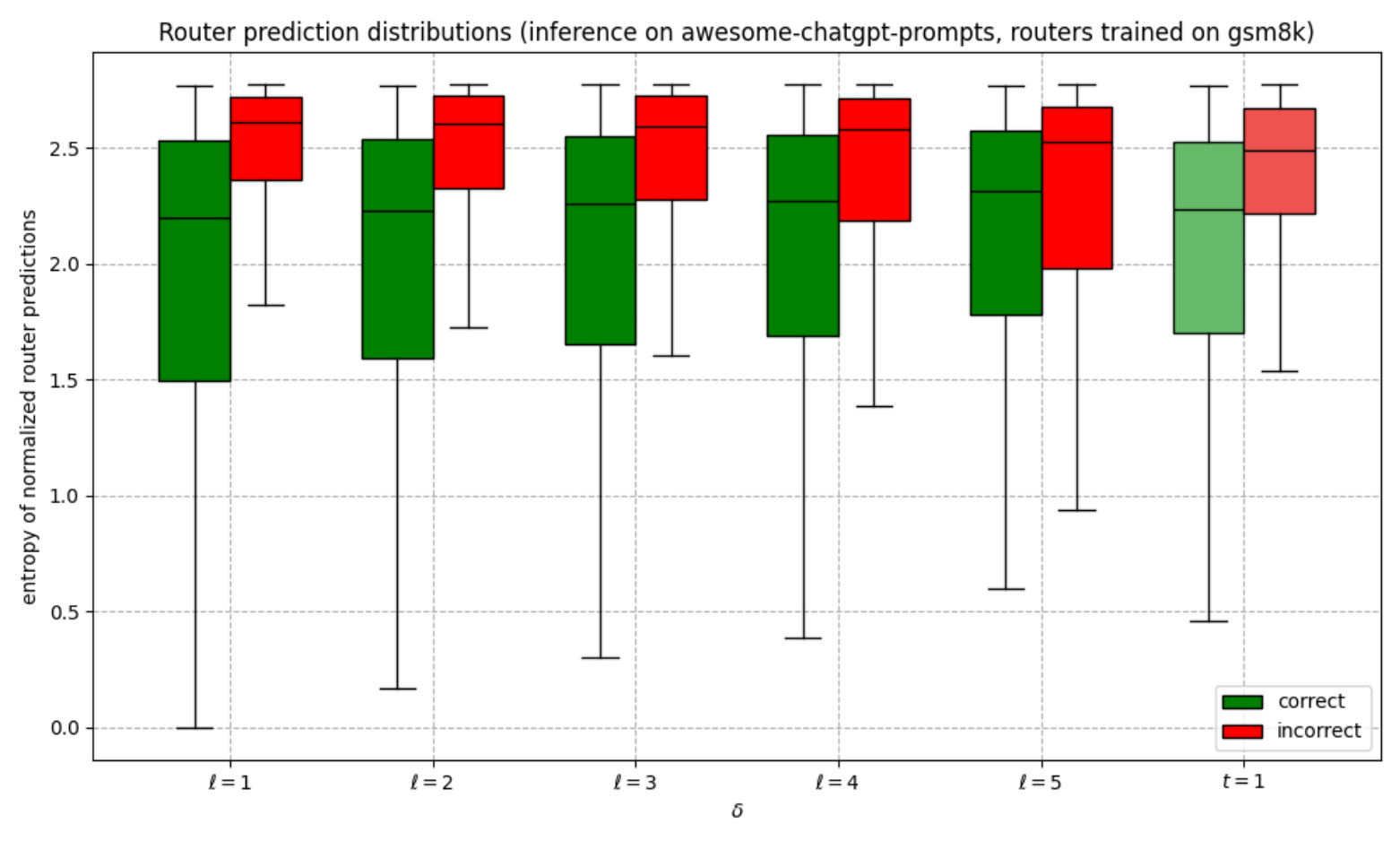
Cache miss rate

0.32	0.13	0.17	0.2	0.22	0.23
0.32					

Prefetch rate

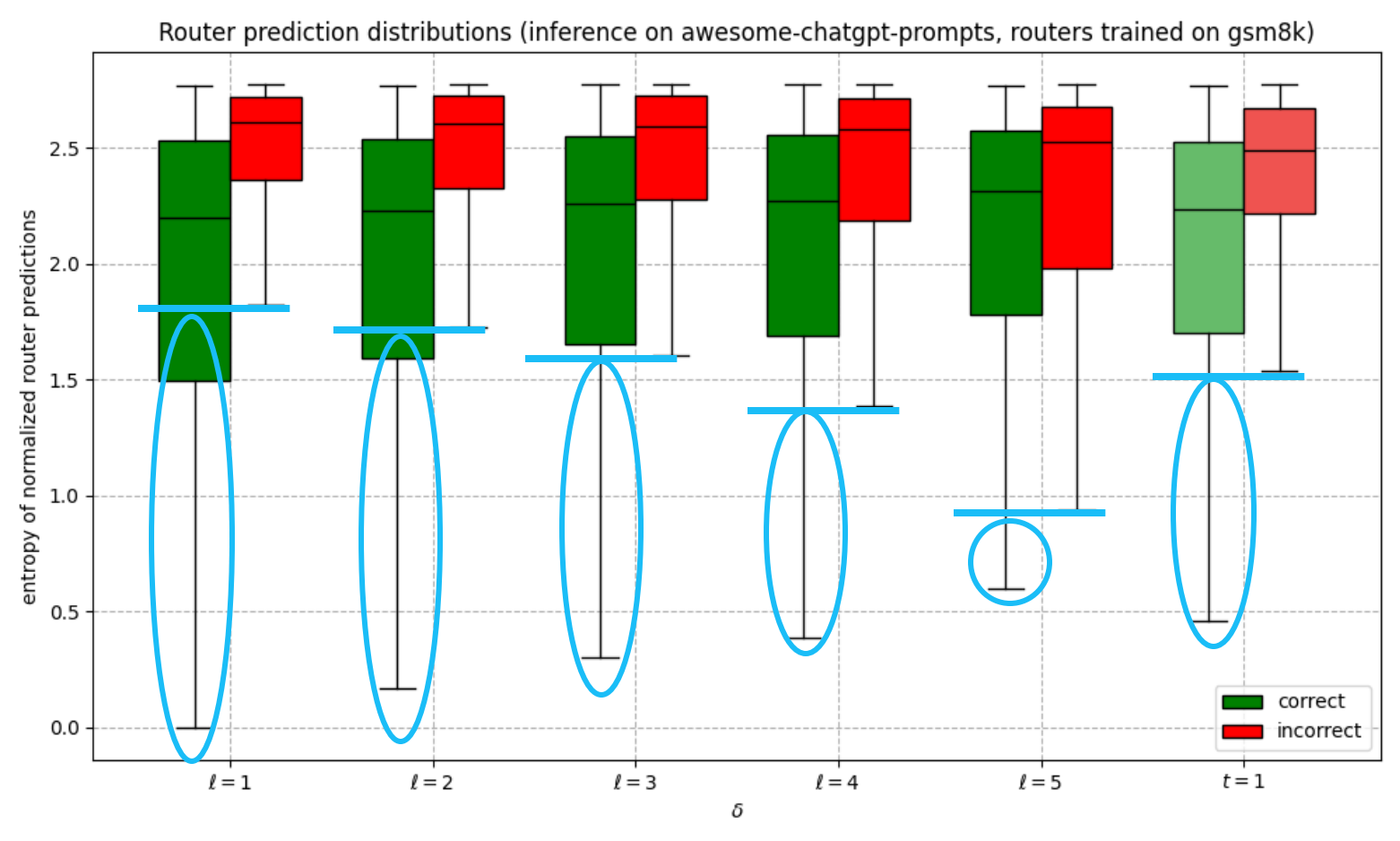
0.36	0.38	0.41	0.41	0.43
0.31	0.31	0.31	0.31	0.31

Assessing Router Confidence: Entropy



Observation: correct predictions have **lower entropy** than incorrect ones

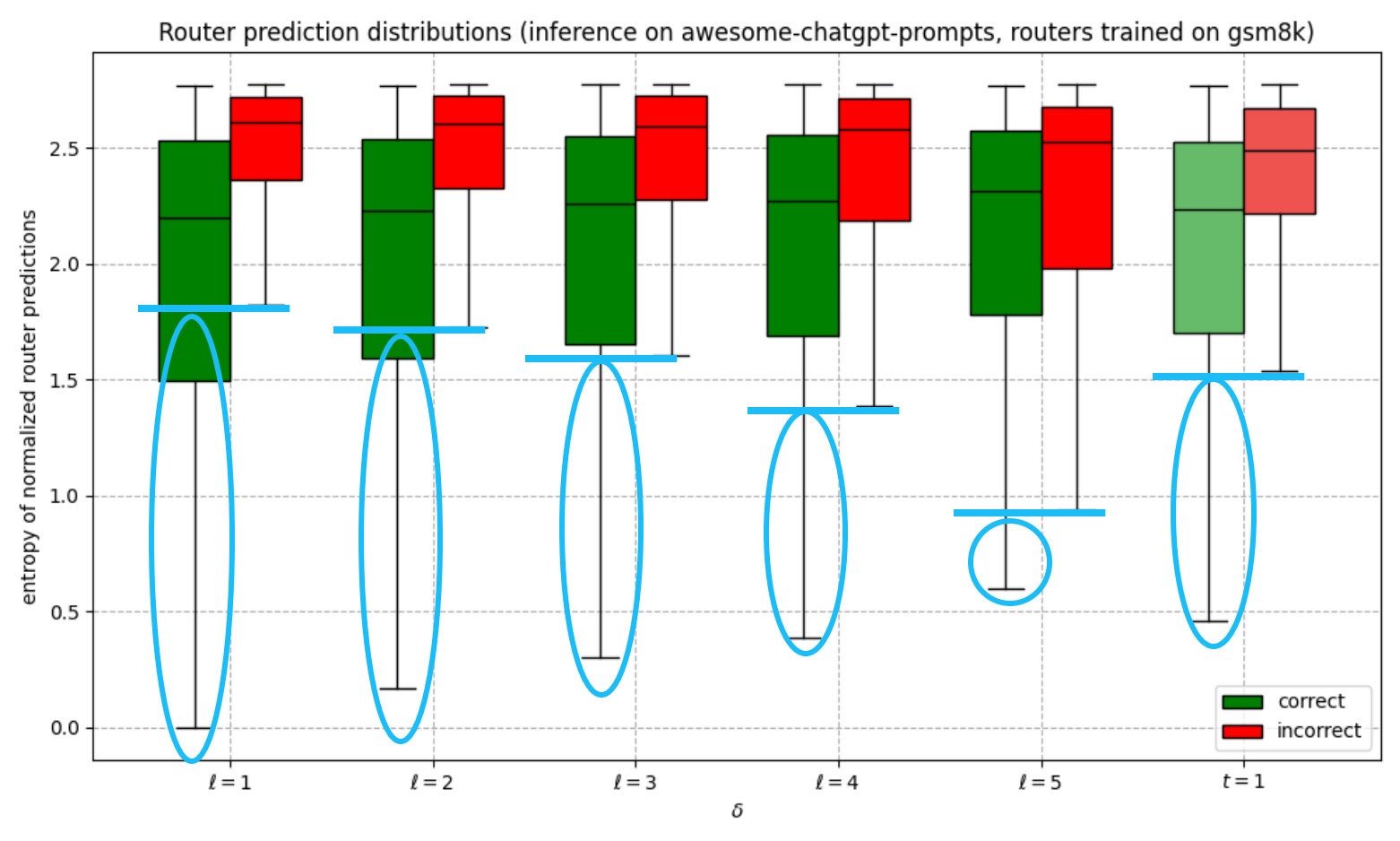
Assessing Router Confidence: Entropy



Observation: correct predictions have **lower entropy** than incorrect ones

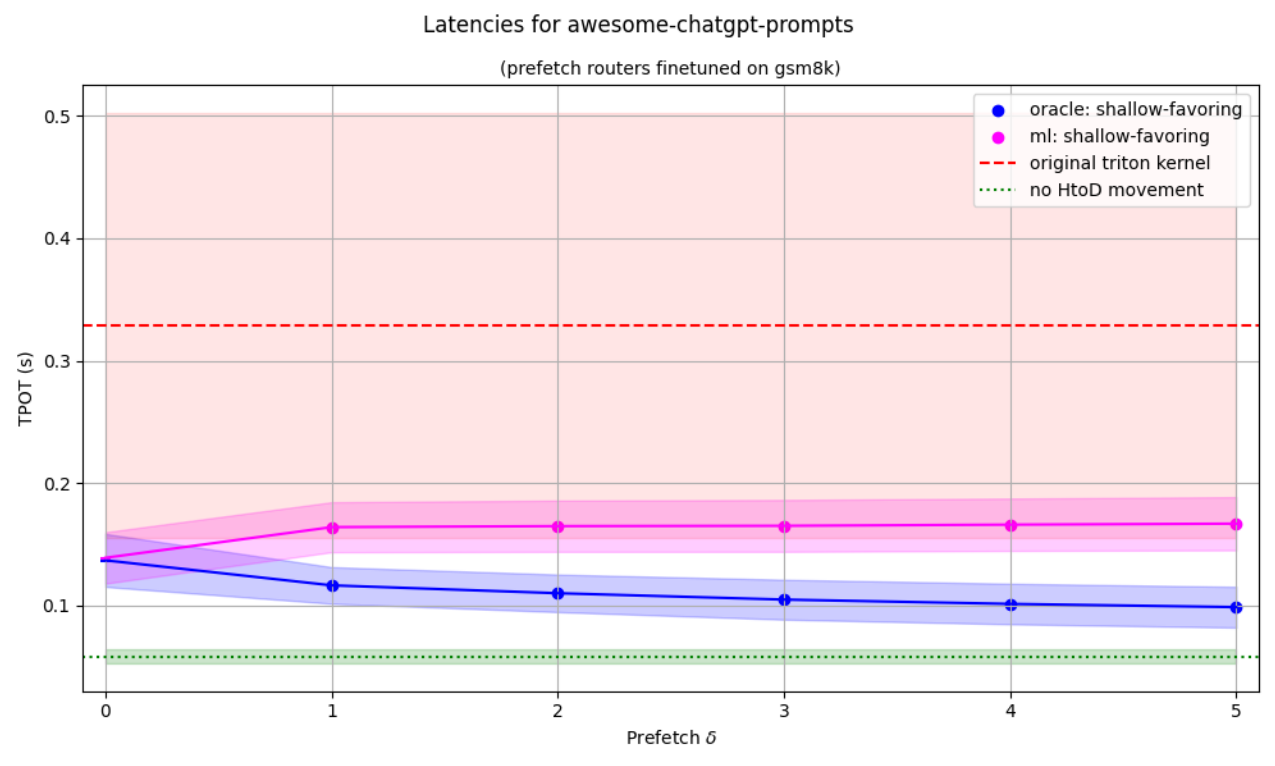
Assessing Router Confidence: Entropy

These thresholds can be calibrated online!



Observation: correct predictions have **lower entropy** than incorrect ones

“Shallow-Favoring” with Entropy Thresholding



Cache miss rate

0.32	0.26	0.28	0.29	0.3	0.31
0.32					

Prefetch rate

0.08	0.07	0.06	0.07	0.05
0.31	0.31	0.31	0.31	0.31

Discussion + Conclusion

Discussion

Discussion

- Oracle expert prefetching provides **realistic lower bound**, and reveals that **prefetching more layers in advance** (up to 5) enables maximum copy-compute overlap (since 1 expert transfer ≅ 4 layers compute)

Discussion

- Oracle expert prefetching provides **realistic lower bound**, and reveals that **prefetching more layers in advance** (up to 5) enables maximum copy-compute overlap (since 1 expert transfer ≅ 4 layers compute)

- MoE inference with offloading is severely memory-bound — critical to minimize:

Discussion

- Oracle expert prefetching provides **realistic lower bound**, and reveals that **prefetching more layers in advance** (up to 5) enables maximum copy-compute overlap (since 1 expert transfer ≅ 4 layers compute)

- MoE inference with offloading is severely memory-bound — critical to minimize:
  - ***#prefetches*** ∪ ***#loads-on-demand***

Discussion

- Oracle expert prefetching provides **realistic lower bound**, and reveals that **prefetching more layers in advance** (up to 5) enables maximum copy-compute overlap (since 1 expert transfer ≅ 4 layers compute)

- MoE inference with offloading is severely memory-bound — critical to minimize:
  - ***#prefetches*** ∪ ***#loads-on-demand***
  - We achieve the lowest counts for that in the entropy-thresholded setting

Discussion

- Oracle expert prefetching provides **realistic lower bound**, and reveals that **prefetching more layers in advance** (up to 5) enables maximum copy-compute overlap (since 1 expert transfer ≅ 4 layers compute)

- MoE inference with offloading is severely memory-bound — critical to minimize:
  - ***#prefetches*** ∪ ***#loads-on-demand***
  - We achieve the lowest counts for that in the entropy-thresholded setting

- We have not yet been able to replicate the downward Oracle trend yet with our ML models — hypotheses:

Discussion

- Oracle expert prefetching provides **realistic lower bound**, and reveals that **prefetching more layers in advance** (up to 5) enables maximum copy-compute overlap (since 1 expert transfer ≅ 4 layers compute)

- MoE inference with offloading is severely memory-bound — critical to minimize:
  - ***#prefetches*** ∪ ***#loads-on-demand***
  - We achieve the lowest counts for that in the entropy-thresholded setting

- We have not yet been able to replicate the downward Oracle trend yet with our ML models — hypotheses:
  - For skip-layer router prediction, **predictive accuracy deteriorates** as δ increases — in entropy-thresholded case, this amounts to *too little/not enough prefetching* for good overlap

Discussion

- Oracle expert prefetching provides **realistic lower bound**, and reveals that **prefetching more layers in advance** (up to 5) enables maximum copy-compute overlap (since 1 expert transfer ≅ 4 layers compute)

- MoE inference with offloading is severely memory-bound — critical to minimize:
  - ***#prefetches*** ∪ ***#loads-on-demand***
  - We achieve the lowest counts for that in the entropy-thresholded setting

- We have not yet been able to replicate the downward Oracle trend yet with our ML models — hypotheses:
  - For skip-layer router prediction, **predictive accuracy deteriorates** as δ increases — in entropy-thresholded case, this amounts to *too little/not enough prefetching* for good overlap
  - **Running the extra routers** contributes non-negligible TPOT latency

Discussion

- Oracle expert prefetching provides **realistic lower bound**, and reveals that **prefetching more layers in advance** (up to 5) enables maximum copy-compute overlap (since 1 expert transfer ≅ 4 layers compute)

- MoE inference with offloading is severely memory-bound — critical to minimize:
  - ***#prefetches*** ∪ ***#loads-on-demand***
  - We achieve the lowest counts for that in the entropy-thresholded setting

- We have not yet been able to replicate the downward Oracle trend yet with our ML models — hypotheses:
  - For skip-layer router prediction, **predictive accuracy deteriorates** as δ increases — in entropy-thresholded case, this amounts to *too little/not enough prefetching* for good overlap
  - **Running the extra routers** contributes non-negligible TPOT latency
  - **Prefetching** (for ℓ + δ) **and on-demand loading** (for ℓ) **may execute simultaneously**, causing bubbles that just on-demand loading (δ =0) doesn't have

Discussion

- Oracle expert prefetching provides **realistic lower bound**, and reveals that **prefetching more layers in advance** (up to 5) enables maximum copy-compute overlap (since 1 expert transfer ≅ 4 layers compute)

- MoE inference with offloading is severely memory-bound — critical to minimize:
  - ***#prefetches*** ∪ ***#loads-on-demand***
  - We achieve the lowest counts for that in the entropy-thresholded setting

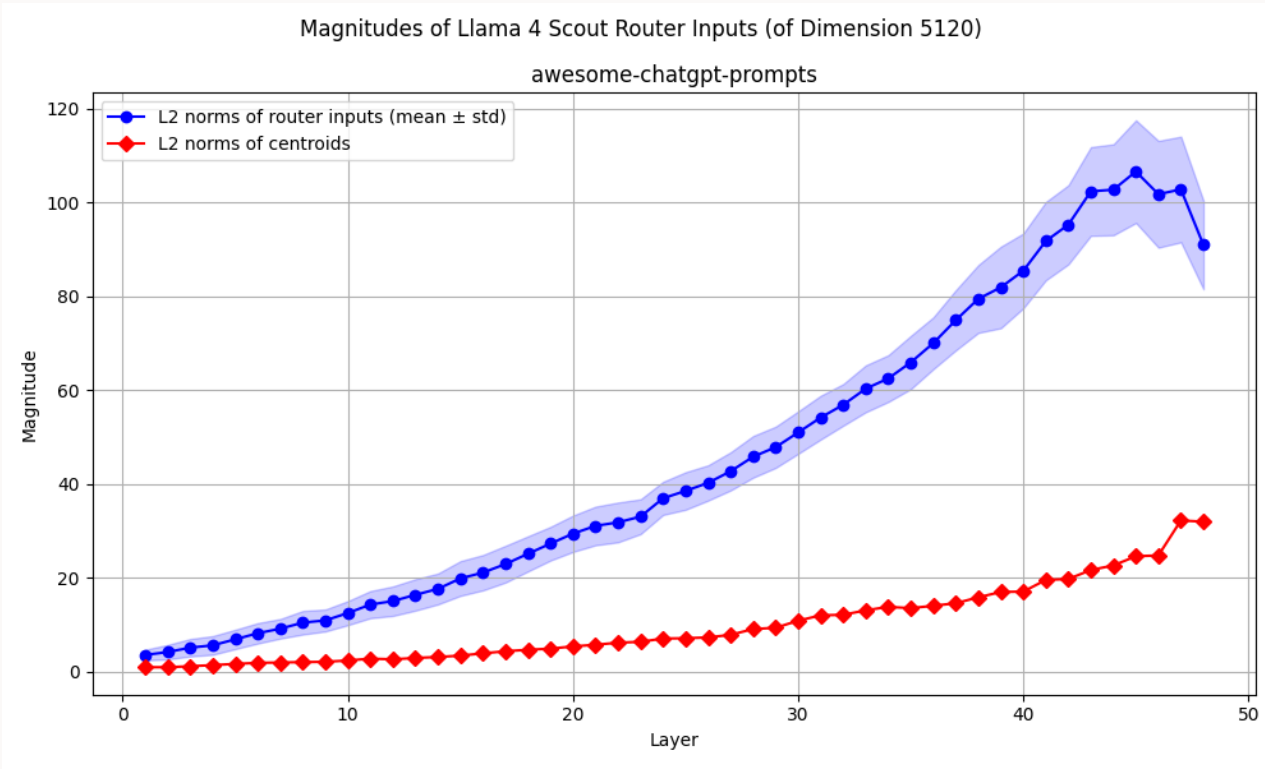
- We have not yet been able to replicate the downward Oracle trend yet with our ML models — hypotheses:
  - For skip-layer router prediction, **predictive accuracy deteriorates** as δ increases — in entropy-thresholded case, this amounts to *too little/not enough prefetching* for good overlap
  - **Running the extra routers** contributes non-negligible TPOT latency
  - **Prefetching** (for ℓ + δ) **and on-demand loading** (for ℓ) **may execute simultaneously**, causing bubbles that just on-demand loading (δ =0) doesn't have

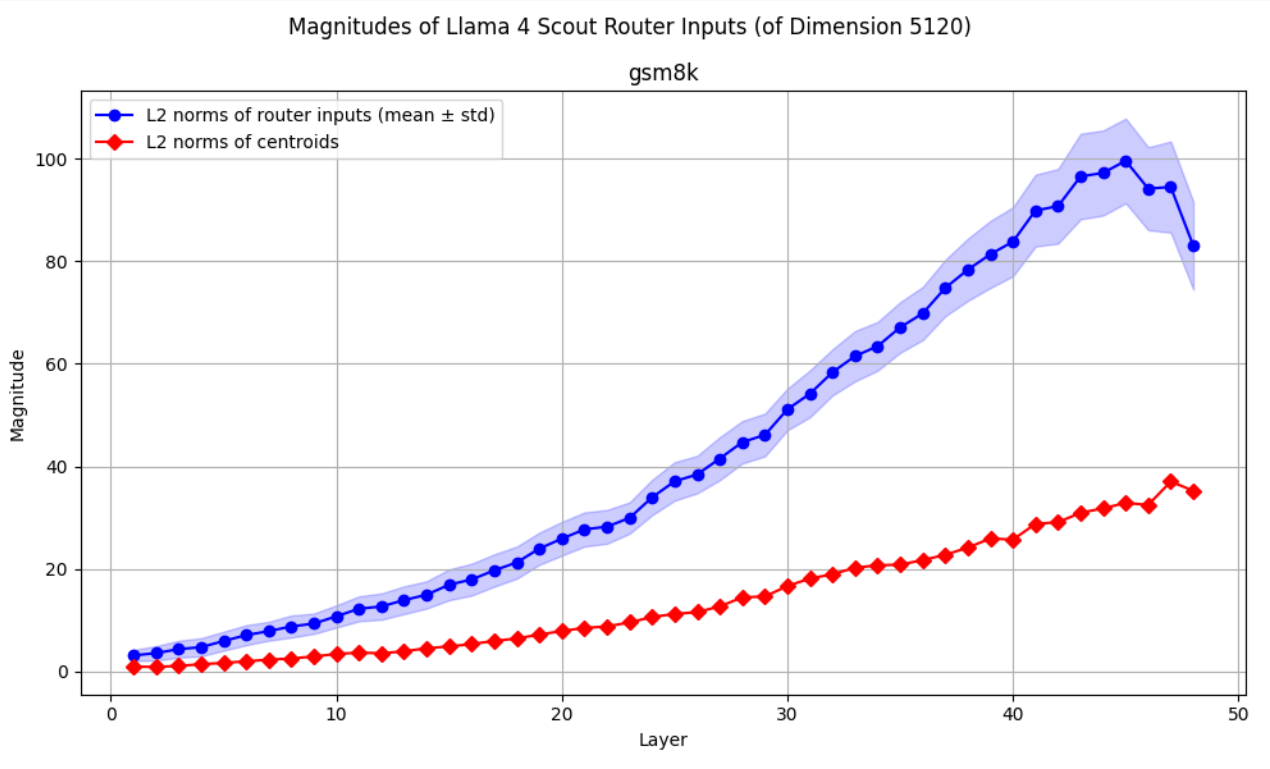
- For future work, work on translating improvements in cache/prefetching stats to desired downward ML trend

Future Directions / WIP

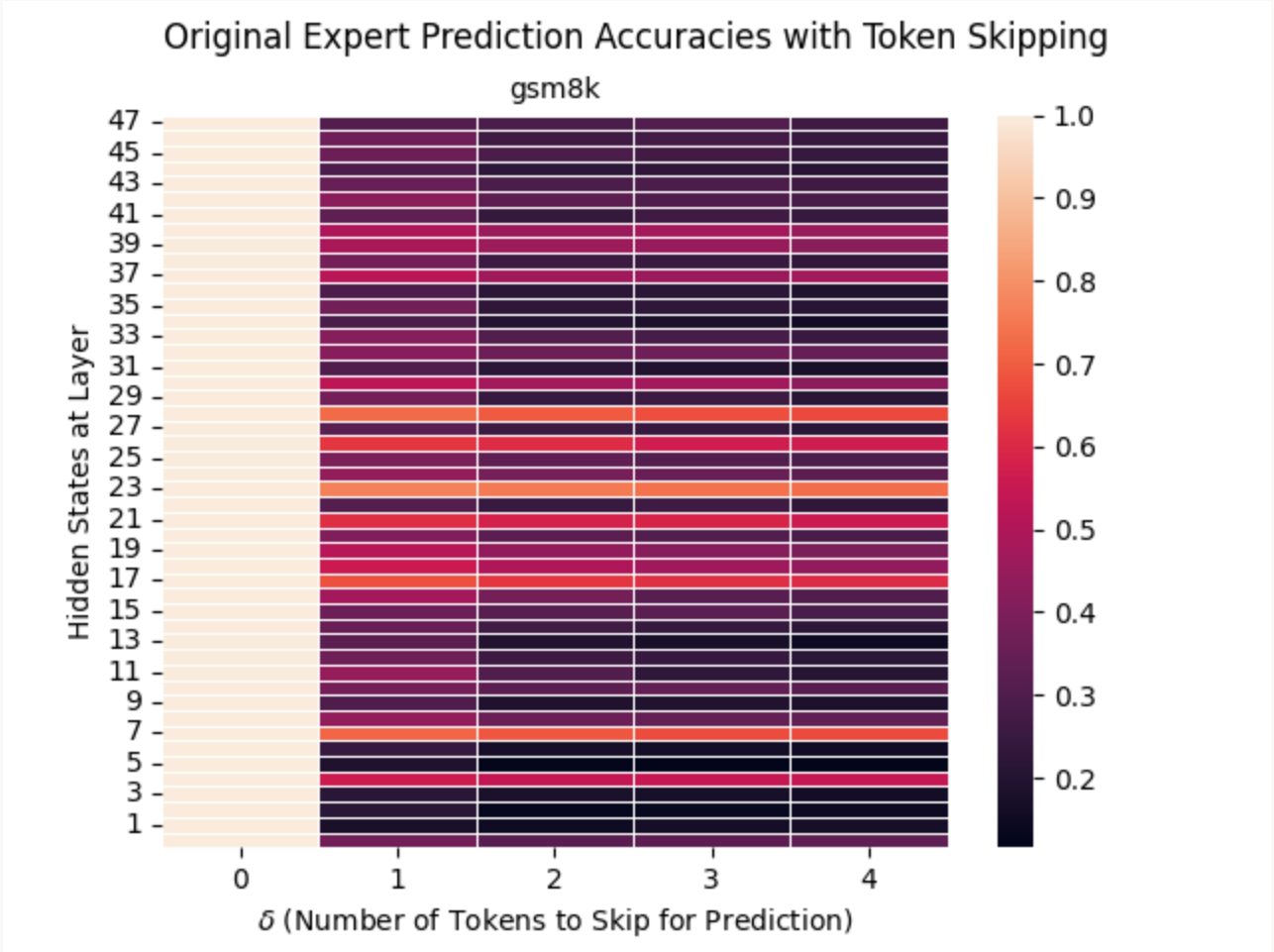
Expert Prediction and Residual Stream

- Work on ML component for skip-layer and skip-token expert prediction — may need novel architecture
- Think about possible connections between residual stream and bias initialization for skip-layer routers

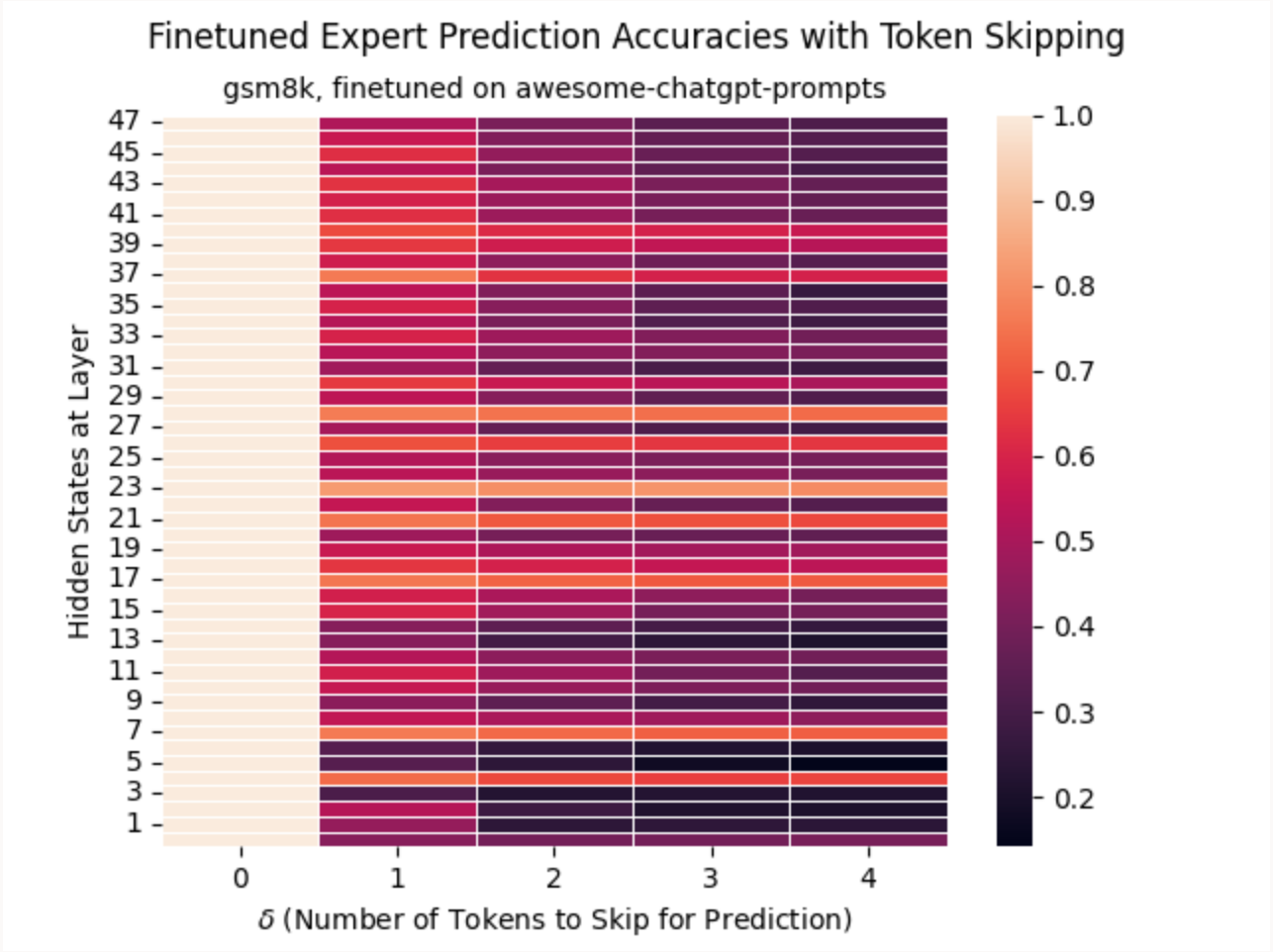




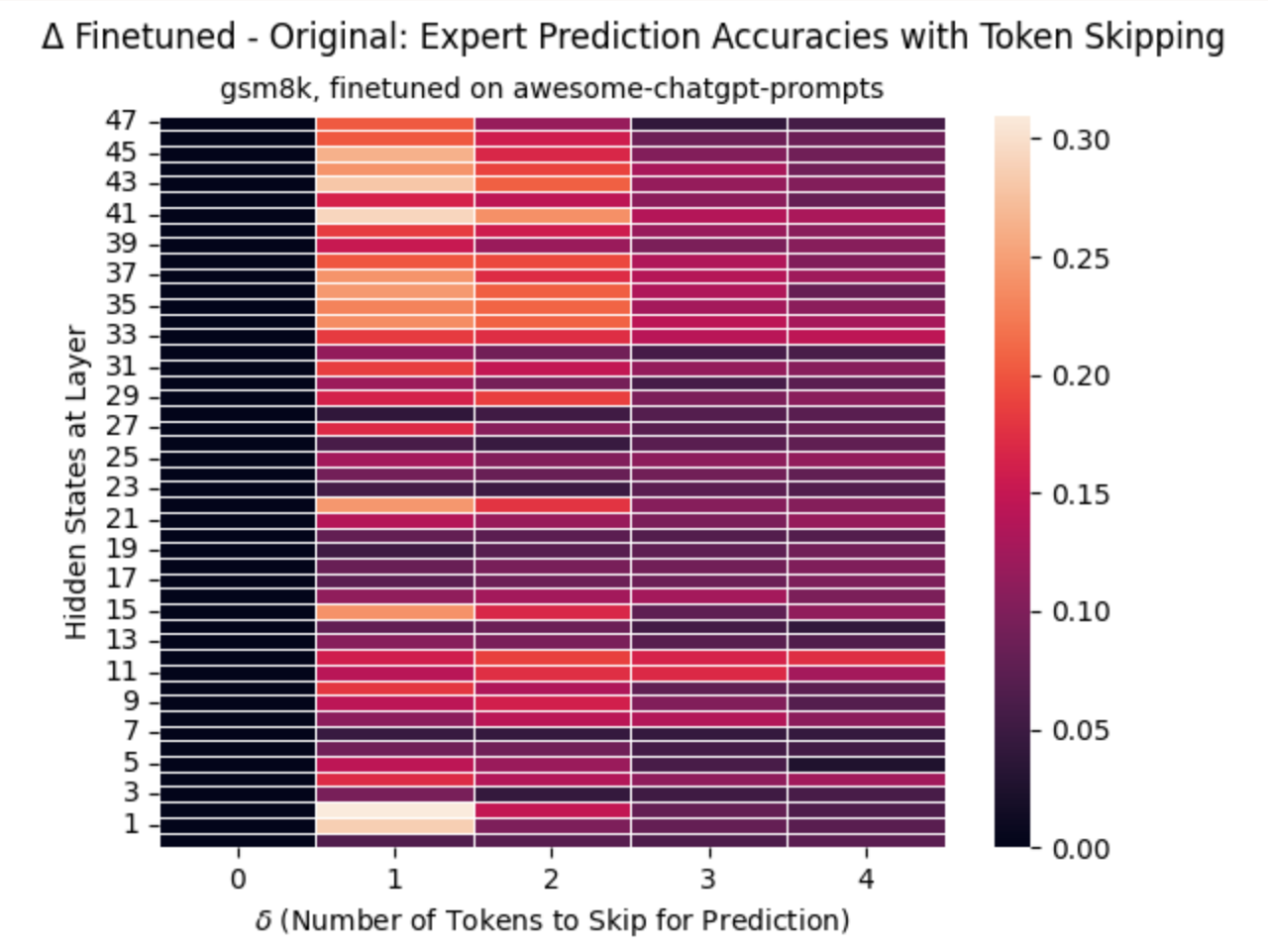
Llama4Scout: Expert Prediction with Token Skipping (Original Routers)



Llama4Scout: Expert Prediction with Token Skipping (Finetuned Routers)



Llama4Scout: Expert Prediction with Token Skipping (Improvement)



Stuff/Lessons Learned

Stuff/Lessons Learned

- Hands-on exposure to:
  - **torch.cuda**: Streams, Events, asynchronous copies, synchronization barriers
  - **vLLM** (v0 and v1)
  - importance of **caching**
  - importance of **copy-compute overlap**
  - Lots **of trade-offs** between predictive “ambition” and predictive capacity

Stuff/Lessons Learned

- Hands-on exposure to:
  - **torch.cuda**: Streams, Events, asynchronous copies, synchronization barriers
  - **vLLM** (v0 and v1)
  - importance of **caching**
  - importance of **copy-compute overlap**
  - Lots **of trade-offs** between predictive “ambition” and predictive capacity

- Make minimal changes to an existing codebase!!! — abusing semantics of existing API to suit your needs is better than reimplementing anything from scratch
  - *[Originally, wanted to modify actual triton kernel]*

Stuff/Lessons Learned

- Hands-on exposure to:
  - **torch.cuda**: Streams, Events, asynchronous copies, synchronization barriers
  - **vLLM** (v0 and v1)
  - importance of **caching**
  - importance of **copy-compute overlap**
  - Lots of **trade-offs** between predictive “ambition” and predictive capacity

- Make minimal changes to an existing codebase!!! — abusing semantics of existing API to suit your needs is better than reimplementing anything from scratch
  - *[Originally, wanted to modify actual triton kernel]*

- Leaky synchronization barriers can cause nightmares
  - *[Discovered this when adding print statements resolved my bug]*

Thank You!


ORACLE