



MULTIMODAL GENERATIVE AI · MASTER AI · UGA / GRENOBLE INP

Introduction to

KV CACHE

Efficient LLM Inference
and Context as Memory

Thomas Palmeira Ferraz
LIG & NAVER LABS Europe

Lecture roadmap

PART 01 **KV cache fundamentals**

PART 02 Efficient LLM inference

PART 03 Context as memory

PART 04 Advanced topics

PART 05 Key takeaways

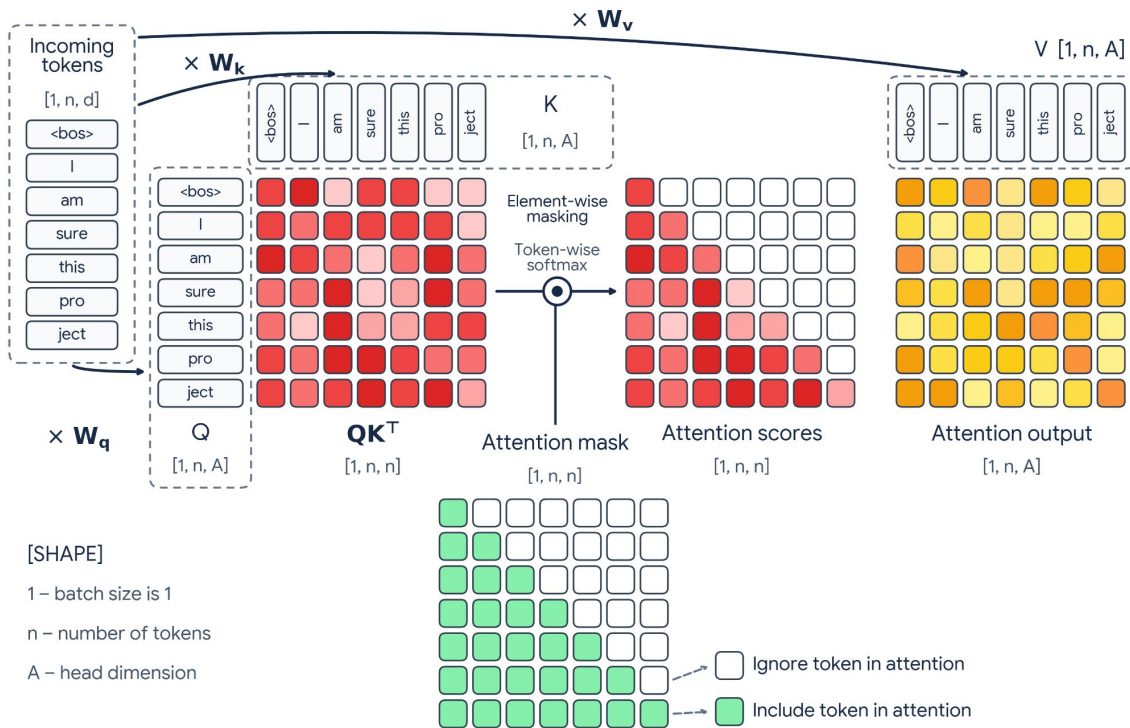
Recap: Causal Self-Attention

$$Q = XW_Q \quad K = XW_K \quad V = XW_V$$

$$\text{Attn}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k} + M)V$$

- Token t attends only to positions $\leq t$.
- Earlier token states cannot depend on later tokens.

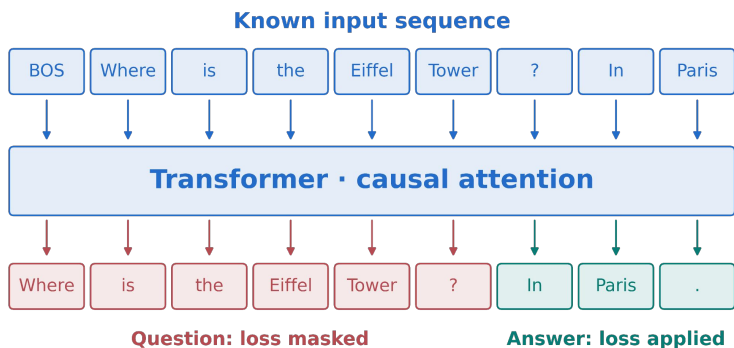
Calculated K/V will not change in the future as new tokens arrive



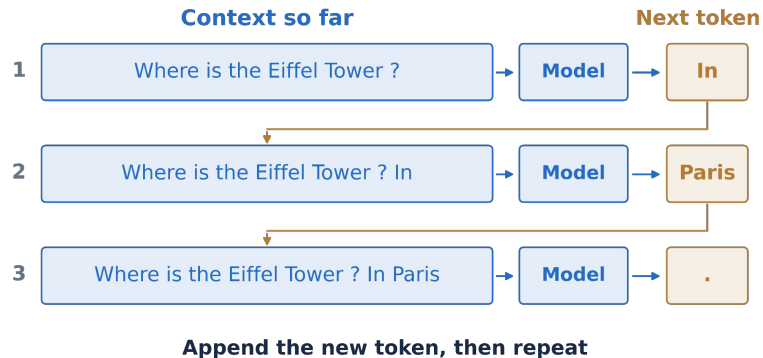
From Training to Autoregressive Inference

Same causal model, different execution patterns.

TRAINING · teacher forcing



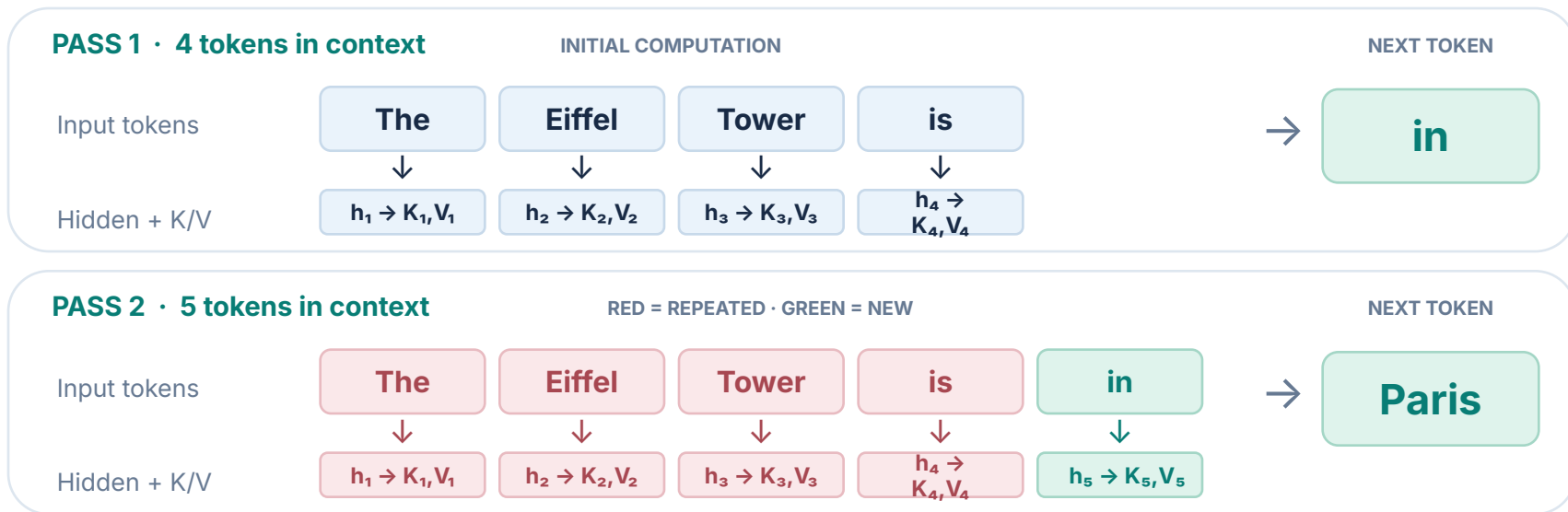
GENERATION · decoding



Causal attention enables parallel training, but not parallel token generation.

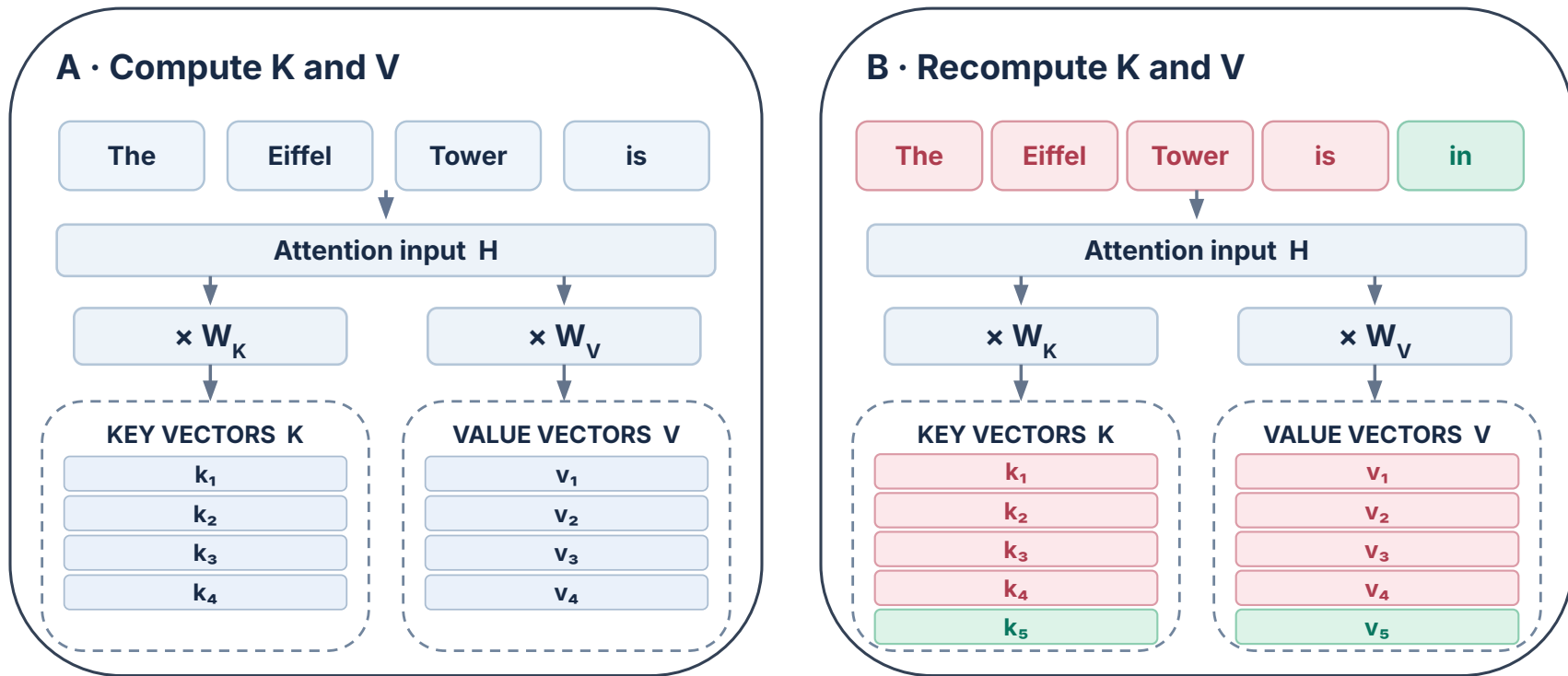
How LLMs Generate Text — Without a Cache

With every new token, the model processes the entire prefix again.



Without caching, old hidden states and K/V are recomputed at every step.

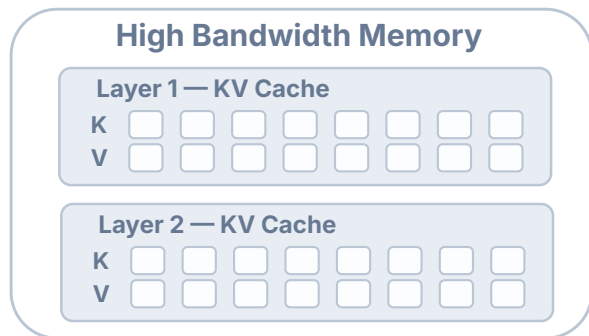
Inside Attention: Recomputing K and V



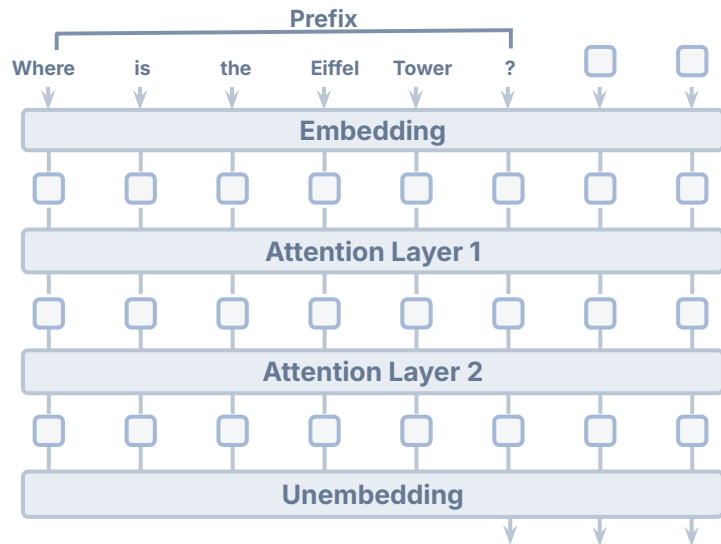
Same K/V, computed twice. Caching could let us reuse the old vectors.

The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Autoregressive Transformer

**1 Architecture**

2 Prefill

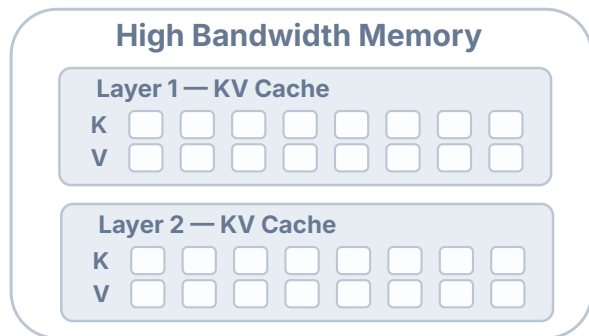
3 Decode #1

4 Decode #2

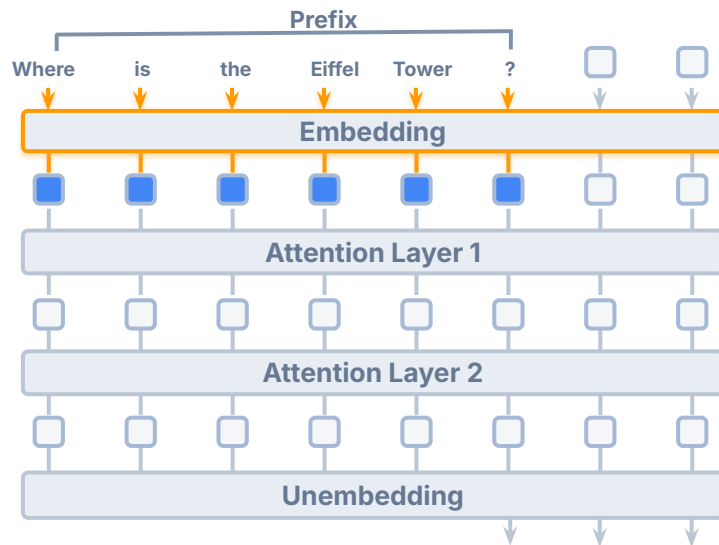
5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Autoregressive Transformer



1 Architecture

2 Prefill

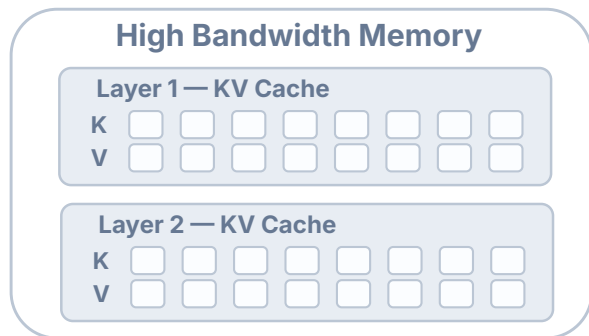
3 Decode #1

4 Decode #2

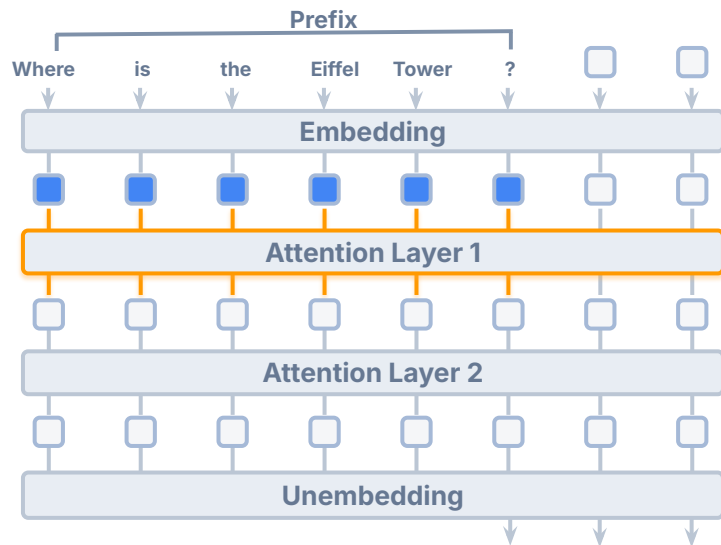
5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Autoregressive Transformer



1 Architecture

2 Prefill

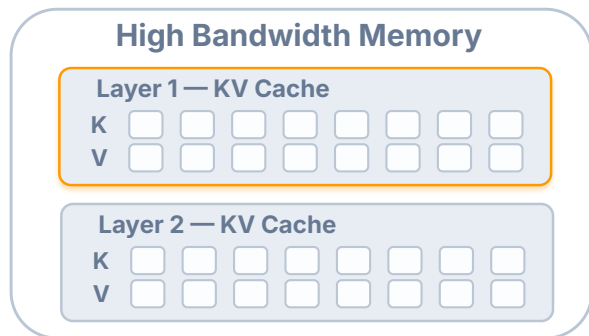
3 Decode #1

4 Decode #2

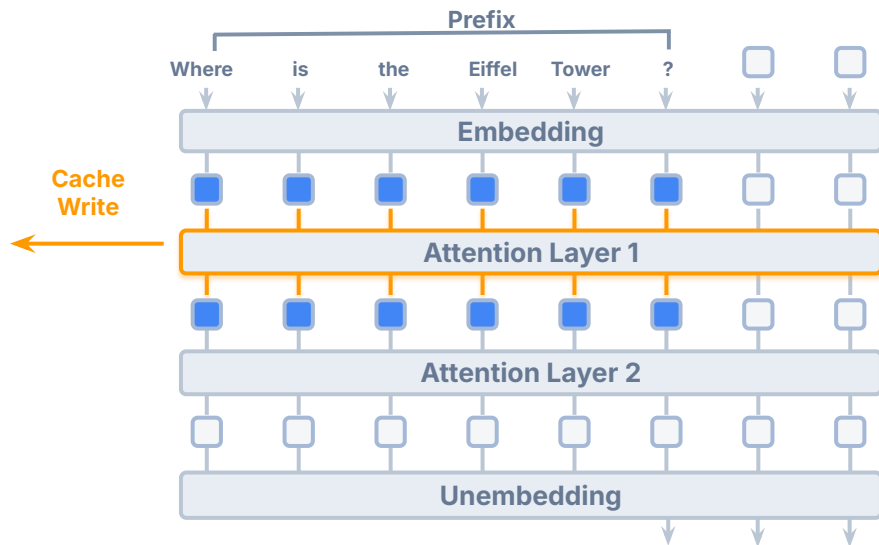
5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Autoregressive Transformer



1 Architecture

2 Prefill

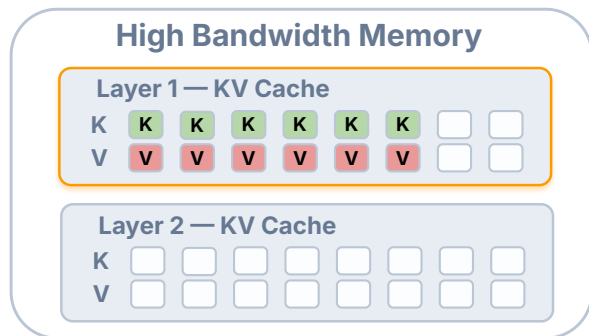
3 Decode #1

4 Decode #2

5 Main idea

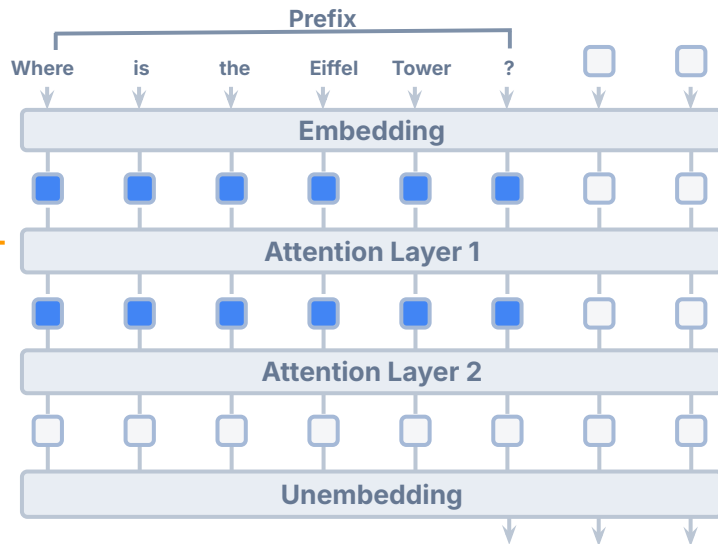
The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Cache
Write

Autoregressive Transformer



1 Architecture

2 Prefill

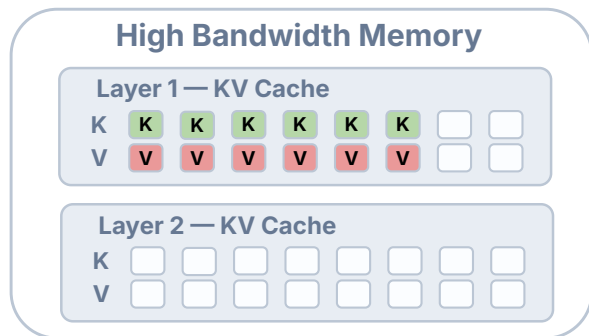
3 Decode #1

4 Decode #2

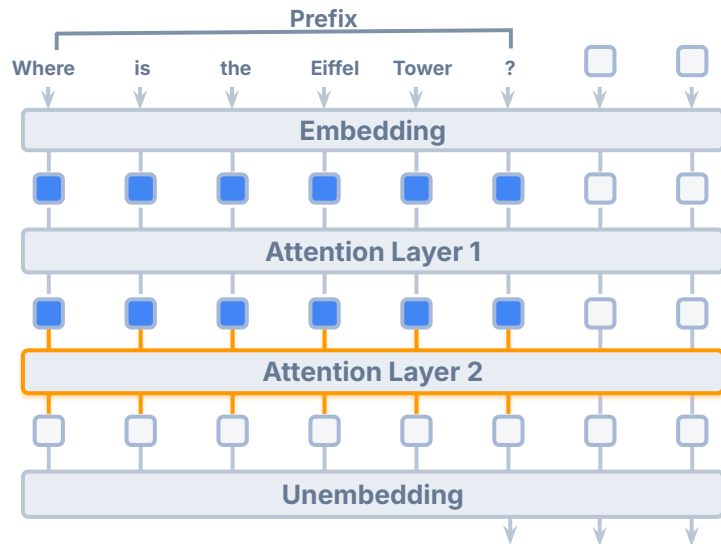
5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Autoregressive Transformer



1 Architecture

2 Prefill

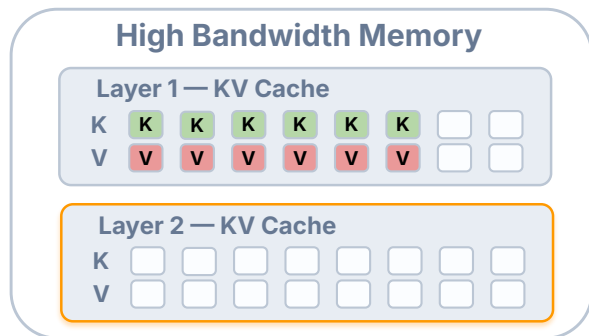
3 Decode #1

4 Decode #2

5 Main idea

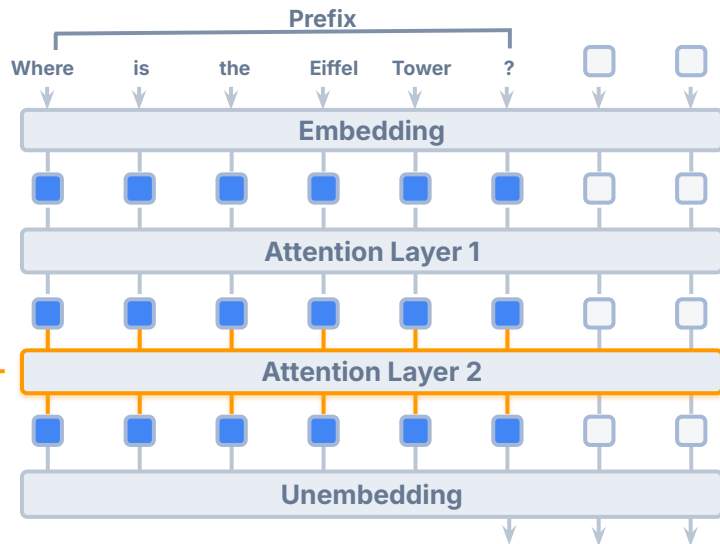
The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Cache
Write

Autoregressive Transformer



1 Architecture

2 Prefill

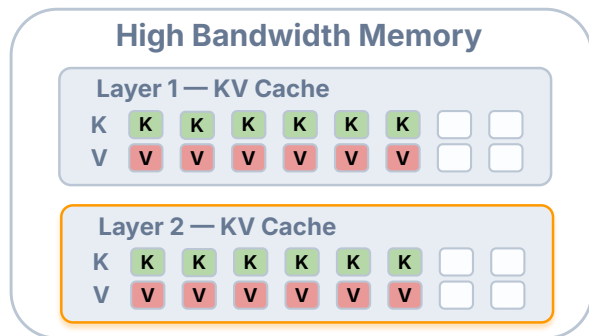
3 Decode #1

4 Decode #2

5 Main idea

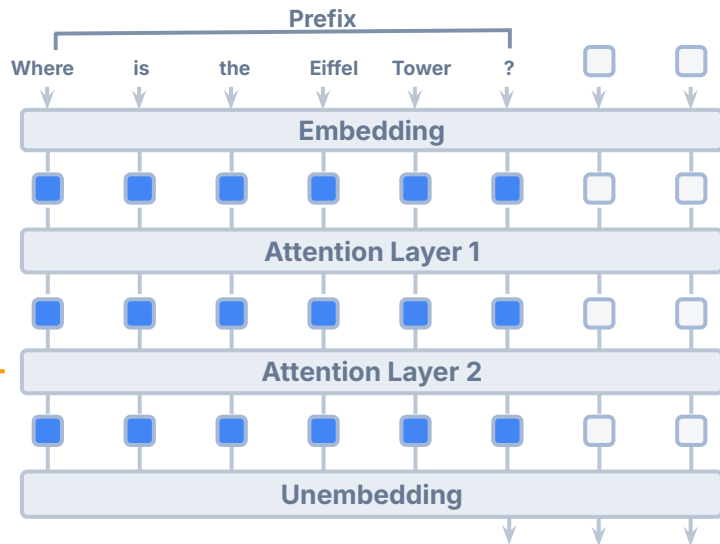
The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Cache
Write

Autoregressive Transformer



1 Architecture

2 Prefill

3 Decode #1

4 Decode #2

5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache

High Bandwidth Memory

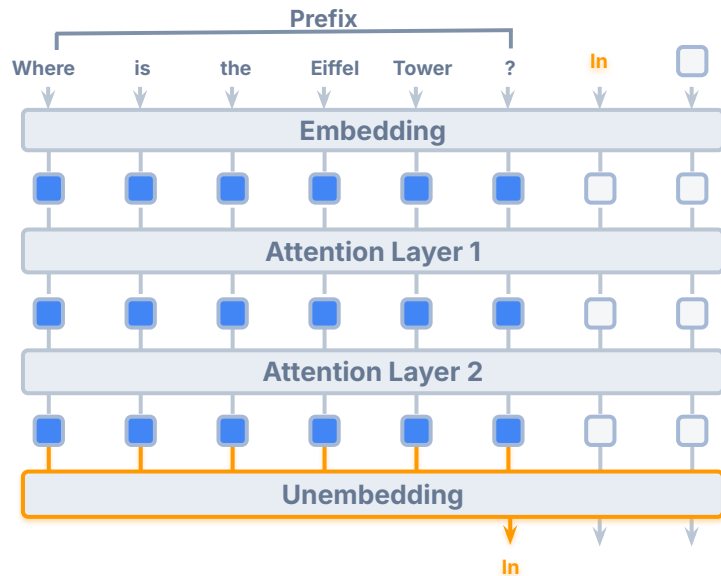
Layer 1 — KV Cache

K	K	K	K	K	K	K		
V	V	V	V	V	V	V		

Layer 2 — KV Cache

K	K	K	K	K	K	K		
V	V	V	V	V	V	V		

Autoregressive Transformer



1 Architecture

2 Prefill

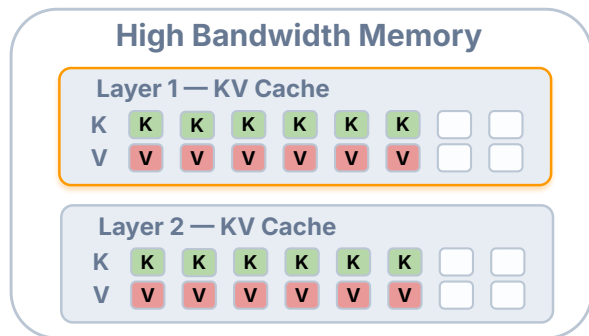
3 Decode #1

4 Decode #2

5 Main idea

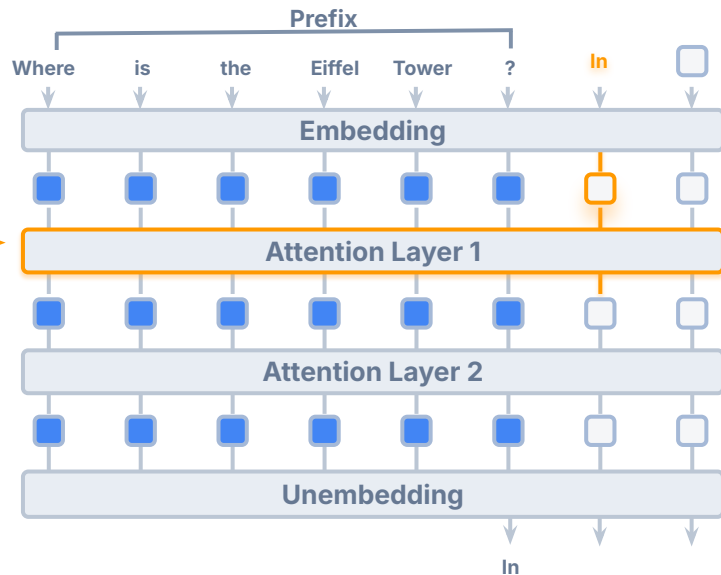
The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Cache
Read

Autoregressive Transformer



1 Architecture

2 Prefill

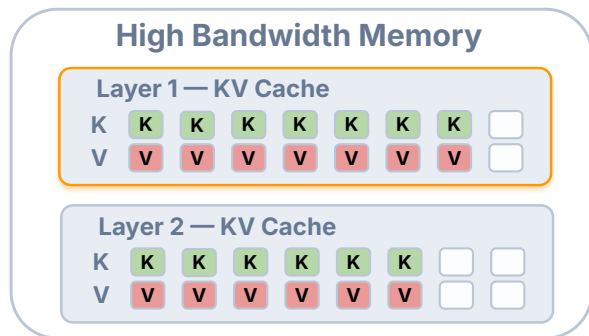
3 Decode #1

4 Decode #2

5 Main idea

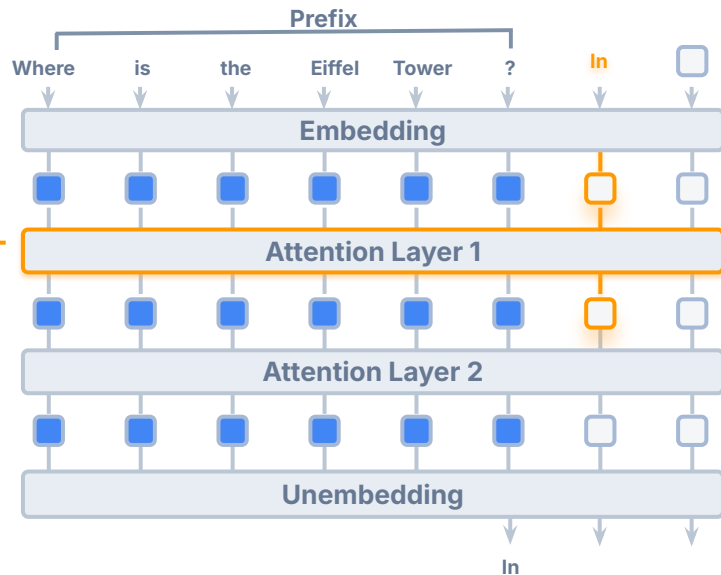
The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Cache
Write

Autoregressive Transformer



1 Architecture

2 Prefill

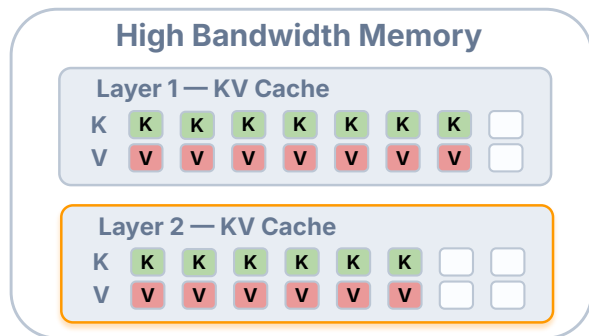
3 Decode #1

4 Decode #2

5 Main idea

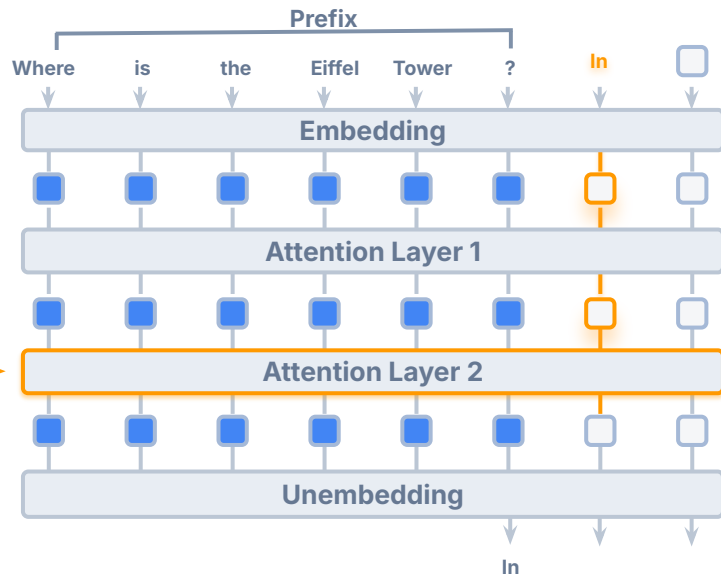
The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Cache
Read

Autoregressive Transformer



1 Architecture

2 Prefill

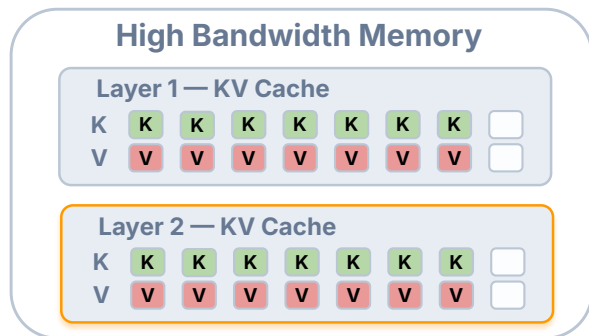
3 Decode #1

4 Decode #2

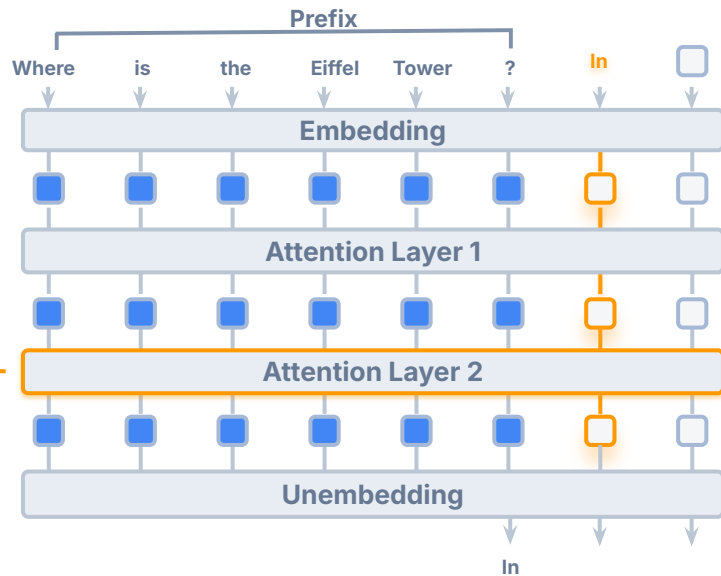
5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache



Autoregressive Transformer



1 Architecture

2 Prefill

3 Decode #1

4 Decode #2

5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache

High Bandwidth Memory

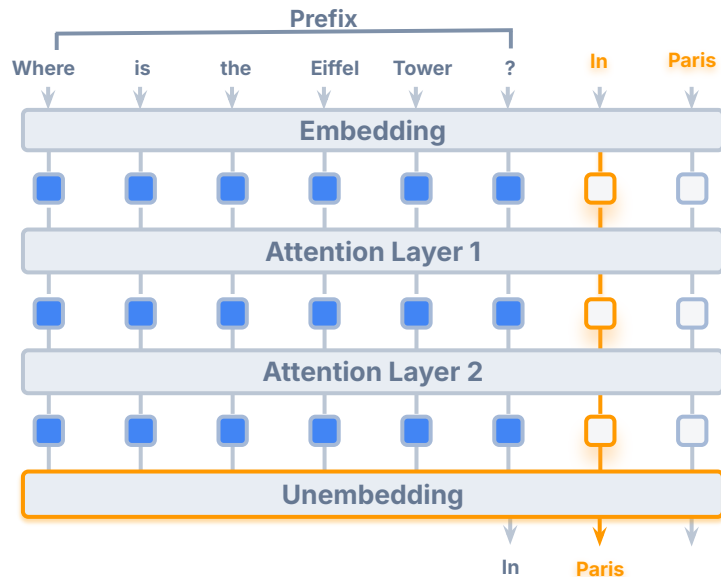
Layer 1 — KV Cache

K	K	K	K	K	K	K	K	
V	V	V	V	V	V	V	V	

Layer 2 — KV Cache

K	K	K	K	K	K	K	K	
V	V	V	V	V	V	V	V	

Autoregressive Transformer



1 Architecture

2 Prefill

3 Decode #1

4 Decode #2

5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache

High Bandwidth Memory

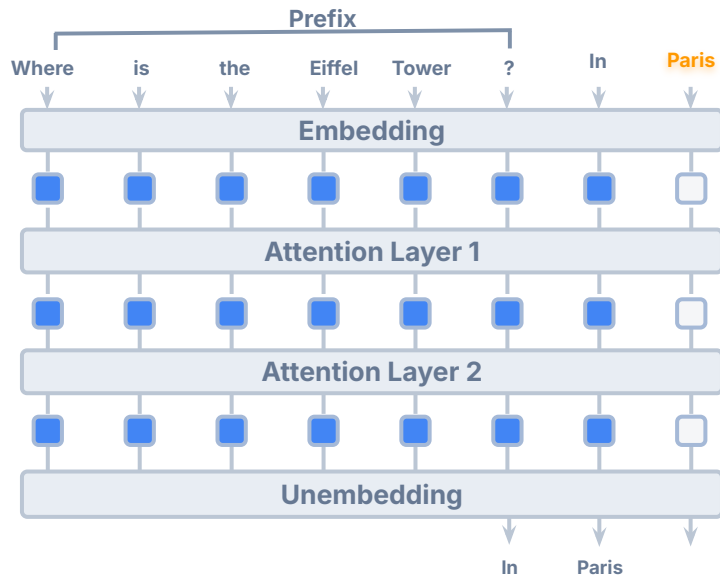
Layer 1 — KV Cache

K	K	K	K	K	K	K	K	
V	V	V	V	V	V	V	V	

Layer 2 — KV Cache

K	K	K	K	K	K	K	K	
V	V	V	V	V	V	V	V	

Autoregressive Transformer



1 Architecture

2 Prefill

3 Decode #1

4 Decode #2

5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache

High Bandwidth Memory

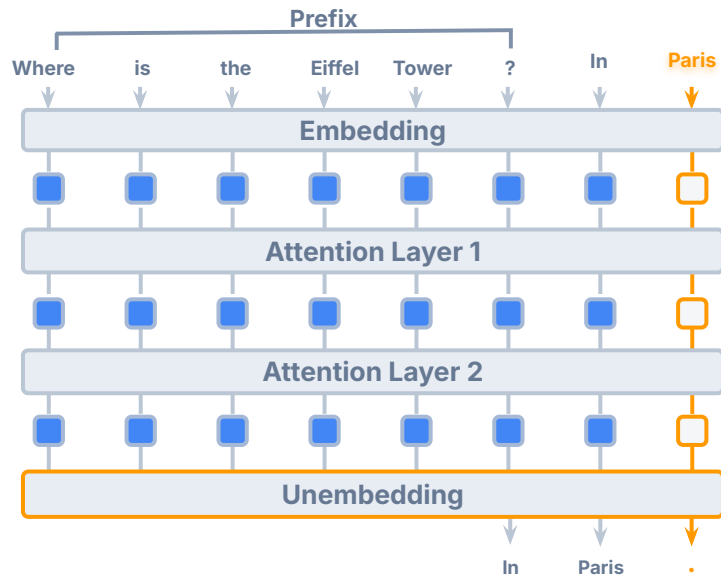
Layer 1 — KV Cache

K	K	K	K	K	K	K	K	K
V	V	V	V	V	V	V	V	V

Layer 2 — KV Cache

K	K	K	K	K	K	K	K	K
V	V	V	V	V	V	V	V	V

Autoregressive Transformer



1 Architecture

2 Prefill

3 Decode #1

4 Decode #2

5 Main idea

The Solution: KV Cache — Visual Walkthrough

Key Value Cache

High Bandwidth Memory

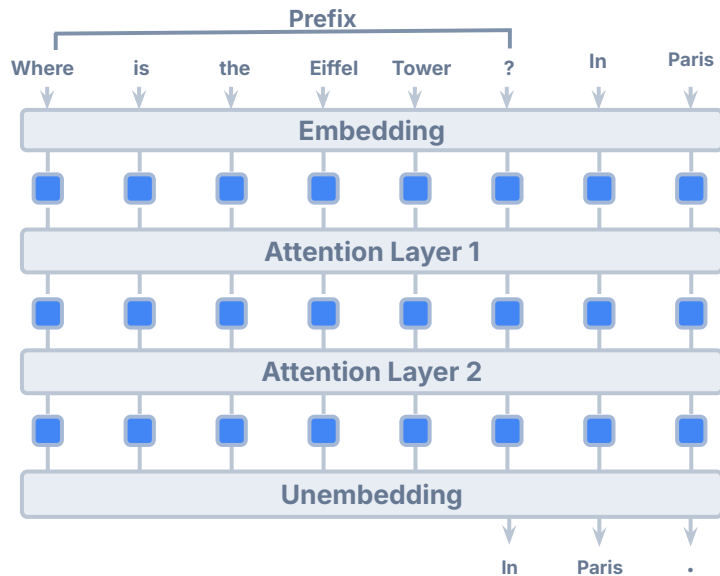
Layer 1 — KV Cache

K	K	K	K	K	K	K	K	K
V	V	V	V	V	V	V	V	V

Layer 2 — KV Cache

K	K	K	K	K	K	K	K	K
V	V	V	V	V	V	V	V	V

Autoregressive Transformer



1 Architecture

2 Prefill

3 Decode #1

4 Decode #2

5 Main idea

Formalizing the Cache

At position t : process only the new layer input h_t (one token embedding).

1. Single-token projections

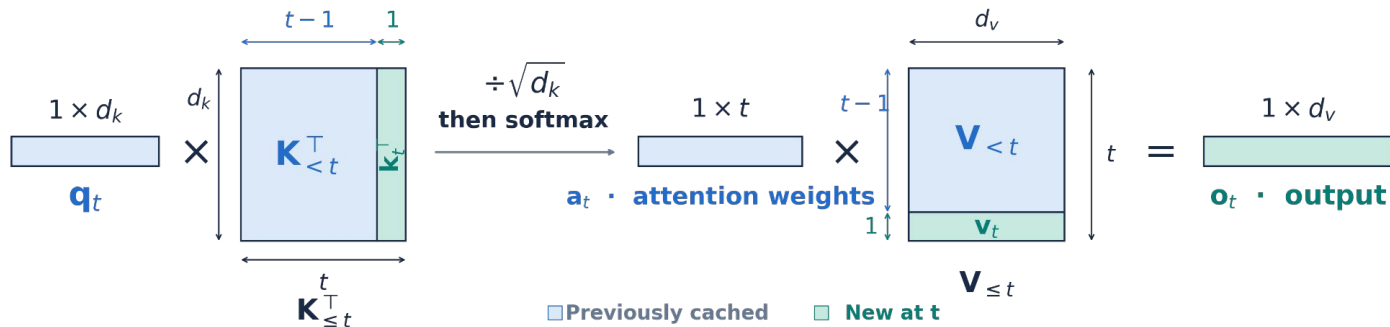
$$\mathbf{q}_t = h_t W_Q, \quad \mathbf{k}_t = h_t W_K, \quad \mathbf{v}_t = h_t W_V$$

2. Append to the cache along the token dimension

$$\mathbf{K}_{\leq t} = [\mathbf{K}_{< t}; \mathbf{k}_t], \quad \mathbf{V}_{\leq t} = [\mathbf{V}_{< t}; \mathbf{v}_t]$$

3. Incremental attention: one query reads all retained K/V

$$\mathbf{o}_t = \text{softmax}(\mathbf{q}_t \mathbf{K}_{\leq t}^T / \sqrt{d_k}) \mathbf{V}_{\leq t}$$



Why not cache Q? Past queries are not needed for future attention outputs.

Why one cache per layer? Each layer produces its own K/V representations.

KV Cache in Code: Recompute vs. Reuse

Example at one attention head at a decoding step: full prefix versus one new token.

WITHOUT CACHE

```
def attention_no_cache(self, X):
    Q = X @ self.W_q
    K = X @ self.W_k
    V = X @ self.W_v

    scores = Q @ K.T / sqrt(d_k)
    scores += causal_mask(len(X))
    weights = softmax(scores, dim=-1)
    return (weights @ V)[-1:]
```



Process one new token

Input: $[t, d] \rightarrow [1, d]$

WITH CACHE

```
def attention_cached(self, x_t):
    q = x_t @ W_q
    k = x_t @ W_k
    v = x_t @ W_v

    K, V = self.kv_cache.append(k, v)
    scores = q @ K.T / sqrt(d_k)
    out = softmax(scores, dim=-1) @ V
    return out
```





Retain previous K/V

K/V storage grows with t

Read the whole history

Scores: $[t, t] \rightarrow [1, t]$

One-token decode needs no causal mask: all cached keys are past/present.
Single head; initialized cache; positions and output projection omitted.

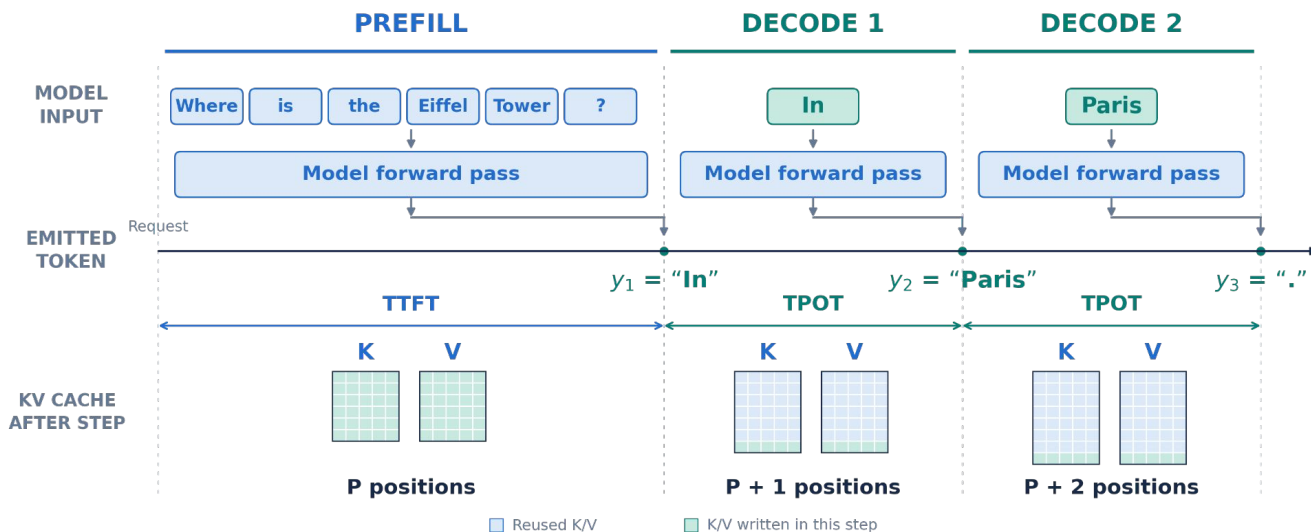
Two Phases of Inference: Prefill and Decode

PREFILL · Prompt processing

Process prompt tokens in parallel, build the initial KV cache, and emit the first token.

DECODE · Incremental generation

Process the last emitted token, read and extend the cache, and emit the next token.



METRICS

TTFT — Time to First Token

Wait from request submission to first output.

TPOT — Time per Output Token

Average interval between output tokens.

KV Cache: Less Computation, More Memory

1 COMPUTATION SAVED (FLOPs)

Without cache → With cache

- At step t : $O(t^2) \rightarrow O(t)$ attention work.
- Across N generated tokens:
 $O(N^3) \rightarrow O(N^2)$ total attention work.



$t \times t$



$1 \times t$

Full-prefix scores

One new query

* Attention FLOPs; fixed model dimensions and prompt length

2 MEMORY REQUIRED

$$M = 2 \times B \times t \times L \times H \times d \times s$$

2: K and V · B: batch · t: cached tokens
L: layers · H: KV heads · d: head dim · s: bytes

Example: Llama 3.1 8B, BF16 weights (≈ 15 GiB)

32 layers · 8 KV heads · $d = 128$ · BF16

8,192 tokens × 128 KiB/token = 1 GiB

Batch size 1: 1 GiB → Batch size 8: 8 GiB

MEASURED SPEEDUP VARIES WITH HARDWARE (Example)

Llama 3, 1B · non-compiled	No cache	KV cache	Speedup
Mac Mini M4 CPU	1 tok/s	68 tok/s	68×
Mac Mini M4 GPU	15 tok/s	62 tok/s	4.1×
NVIDIA A100 GPU	42 tok/s	58 tok/s	1.4×

[Source: S. Raschka](#)

Lecture roadmap

PART 01 KV cache fundamentals

PART 02 **Efficient LLM inference**

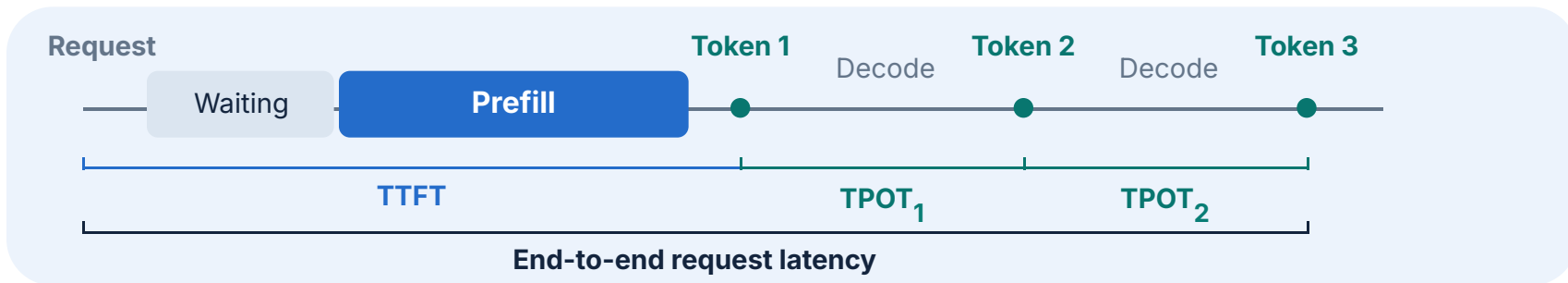
PART 03 Context as memory

PART 04 Advanced topics

PART 05 Key takeaways

What Does “Faster” Mean?

Responsiveness for one user and throughput for the whole system.



Time to First Token (TTFT)

Wait until the first output token.

Time per Output Token (TPOT)

Mean interval after the first token.

Request latency

Wait until the full answer is received.

System throughput

Total output tokens/s across requests.

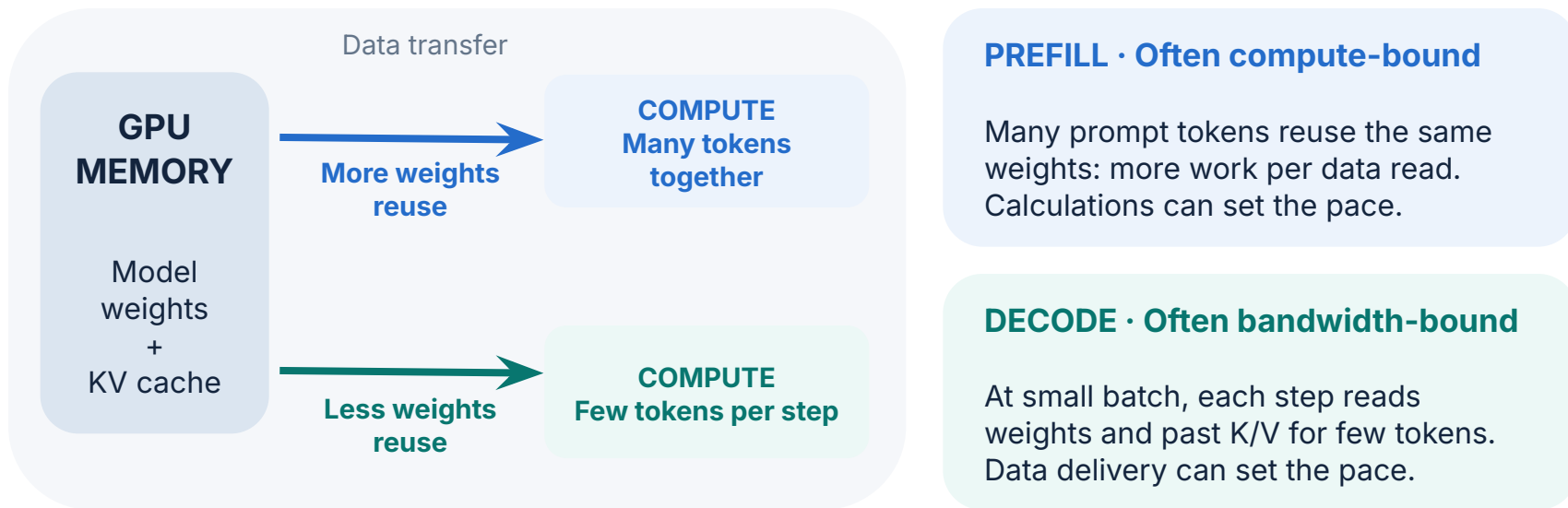
$$\text{Latency} = \text{TTFT} + (N - 1) \times \text{mean TPOT}$$

N output tokens, and N - 1 intervals after the first token.

Higher system throughput can come at the cost of per-request responsiveness.

Why Can the GPU Still Be Waiting?

Speed is limited by both the calculations and how fast data reaches them.

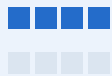


If data arrives too slowly, compute units wait, even when they could calculate faster.

Capacity = how much fits. Bandwidth = how fast it can be read.
The bottleneck depends on batch size, context, model and hardware.

Where Can We Optimize Inference?

Five complementary ways to improve memory use and execution.



Architecture

GQA · MLA

Store fewer / smaller representations



Attention kernels

FlashAttention

Reduce memory traffic



Cache allocation

PagedAttention

Reduce allocation waste



Serving

Batching · prefix caching

Improve utilization and reuse



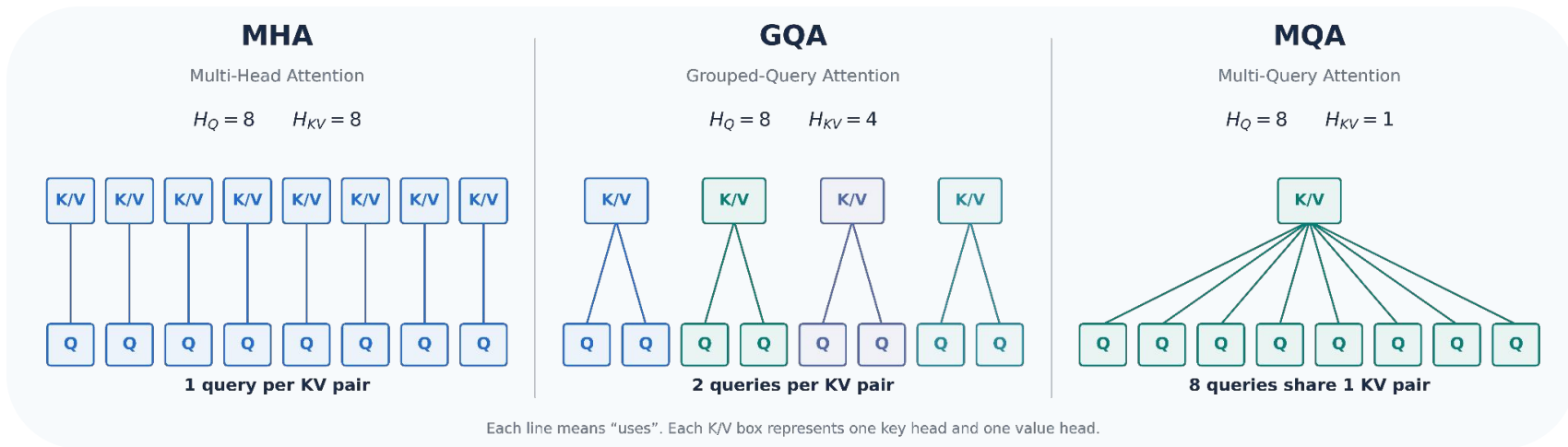
Compression / locality

Quantization · eviction · windowing

Store or attend to less

Sharing K/V Heads: MHA → GQA → MQA

Keep multiple query heads; share their key/value representations.



Fewer KV heads → smaller cache and potentially less memory traffic.

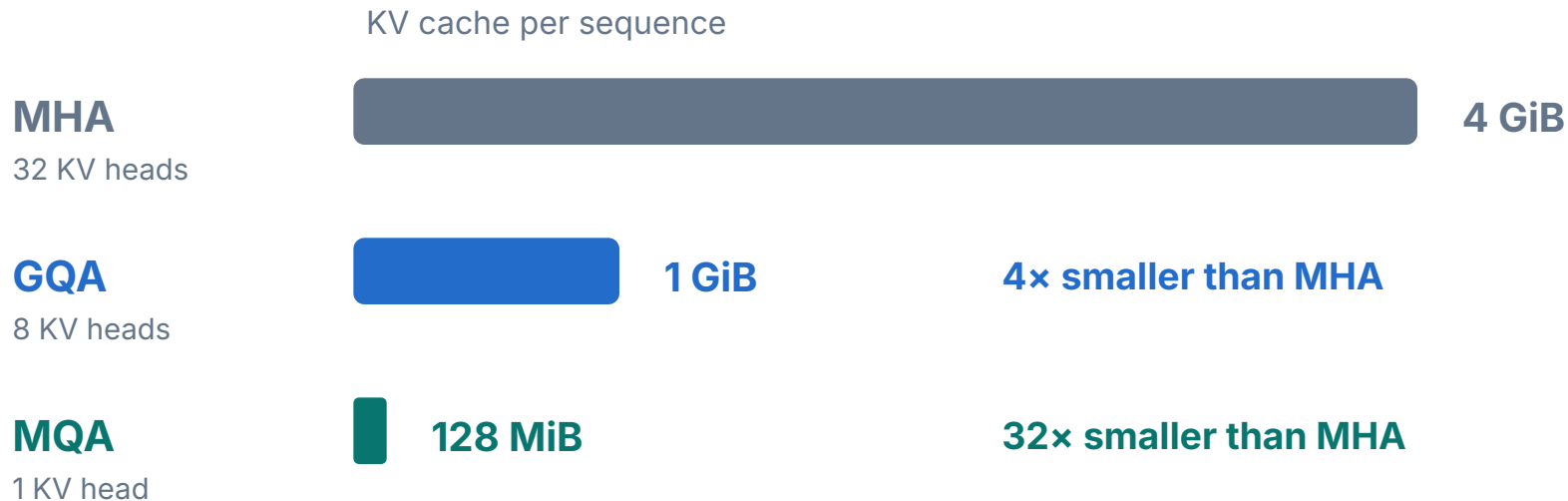
An architectural choice: the model must be trained or adapted for the sharing pattern.

Examples GQA: Llama 3 8B: 32 Q / 8 KV and Qwen2.5 7B: 28 Q / 4 KV

GQA · Ainslie et al. (2023) ↗

How Much KV Memory Does Sharing Save?

Same dimensions; varying only the number of KV heads.



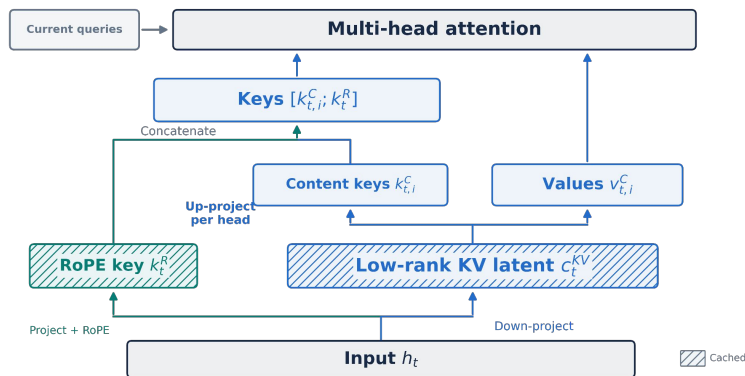
Example: 32 layers · 8,192 cached tokens · head dimension 128 · BF16 · batch 1
GQA matches Llama 3.1 8B. MHA and MQA are hypothetical comparisons,
they are not switches that can be applied to the same trained weights.

(Optional) Beyond GQA: MLA and Cross-Layer KV Reuse

Two architectural approaches: compress each cache or share caches across layers.

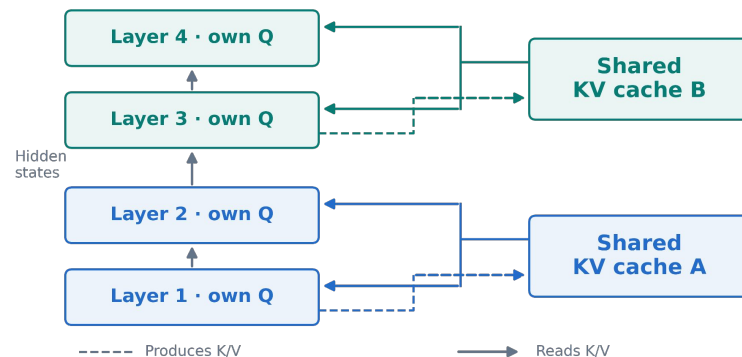
Multi-head Latent Attention

Cache a compact C^{KV} latent plus a separate positional key.



Cross-layer KV Cache Share

Keep layer-specific queries; reuse K/V from a designated source layer.



Architectural support required. Examples: DeepSeek-V2 (MLA); CLA / Gemma 3n (sharing).

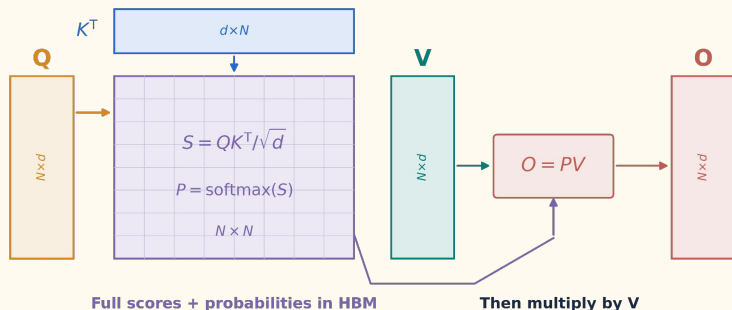
Conceptual MLA computation shown; optimized inference can absorb the up-projections without explicitly reconstructing per-head K/V.

FlashAttention: Compute Attention in Tiles

Tile and combine attention operations to reduce memory traffic.

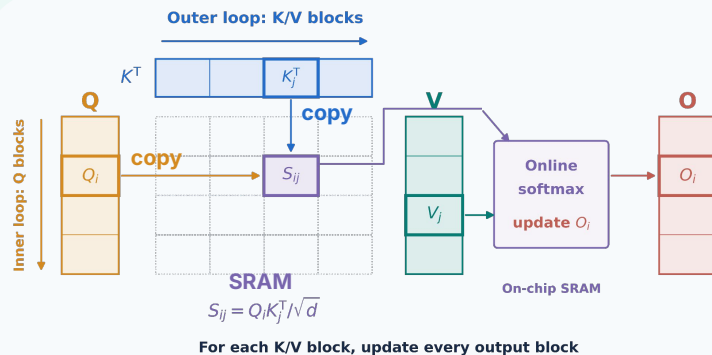
Dao et al. (2022) ↗

Conventional implementation



Write and reread large intermediate score / probability matrices.

FlashAttention



Keep small intermediate tiles on-chip; accumulate the output as you go.

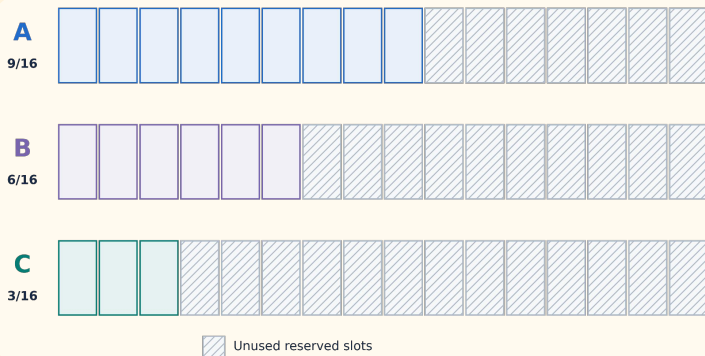
Same dense attention. Less memory traffic and temporary storage.

PagedAttention: Store KV in Flexible Blocks

Allocate blocks as requests grow; preserve token order through a block table.

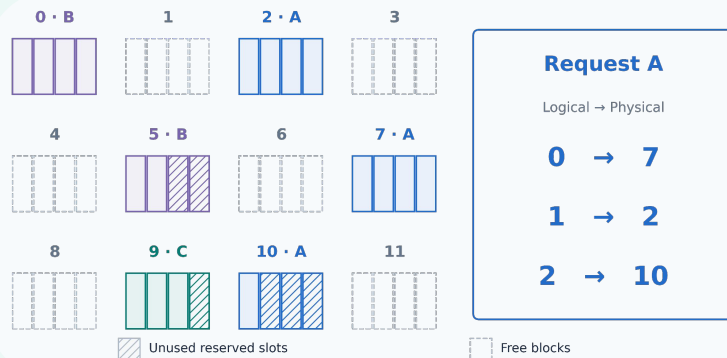
Kwon et al. (2023) ↗

Contiguous reservation



Unused reserved capacity cannot serve other requests.

Paged allocation

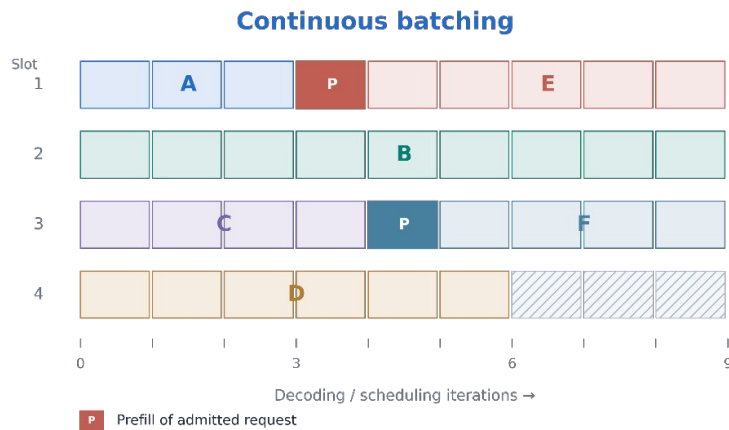
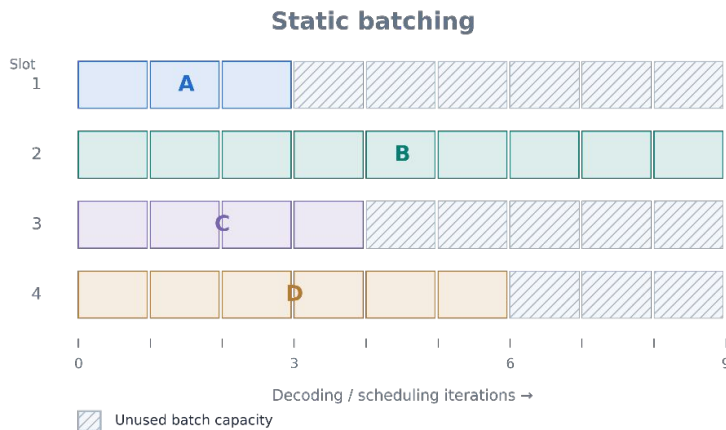


Logical blocks map to scattered physical blocks; free blocks stay available.

Less allocation waste → capacity for more concurrent requests.

Continuous Batching: Serving Multiple Requests

Schedule between iterations to reuse available batch capacity.



STATIC BATCHING

Finished slots wait until the current batch completes.

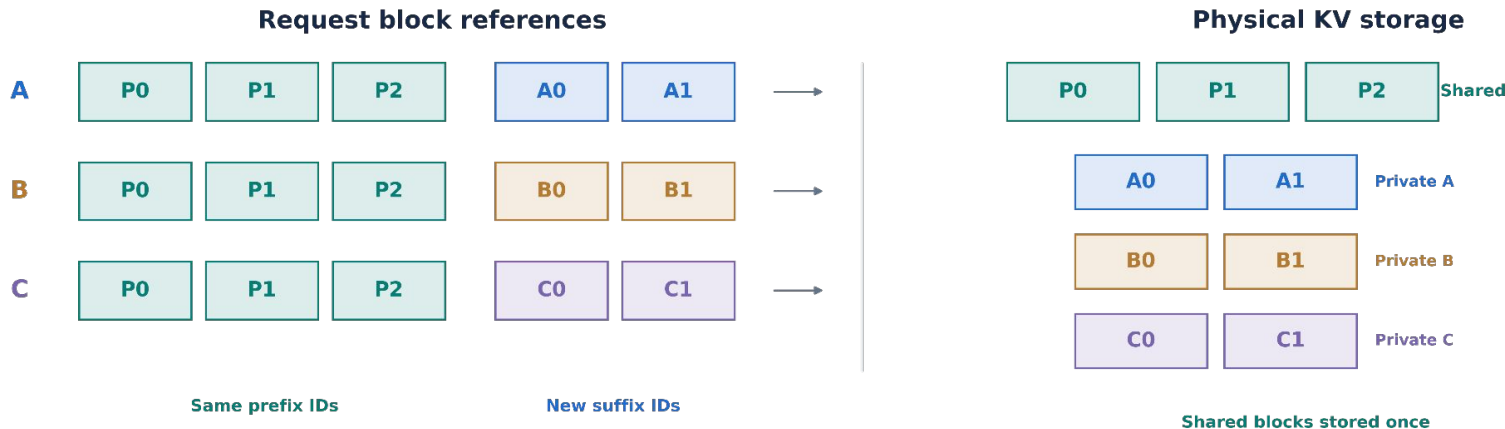
CONTINUOUS BATCHING

Admit waiting requests while others keep decoding.

Initial prefill omitted. P = schematic prefill; iterations are not equal durations. Admission depends on token/KV budgets.

Prefix Caching: Reusing Work Across Requests

Reuse an identical prefix's KV states; prefill only the new suffix.



REUSE MATCHING PREFIX

Skip prefix computation; share its physical KV blocks.

PREFILL NEW SUFFIX

Compute new states while attending to the cached prefix.

Match tokens and model/cache configuration; decode still reads the prefix.

[SGLang / RadixAttention ↗](#)

Practical Ecosystem: Which Tool for Which Job?

Choose by model/hardware support; benchmark TTFT, TPOT, throughput, and KV-memory use.

HF Transformers



**Research /
custom code**

Inspect and modify
KV caches directly

[Documentation ↗](#)

Ollama



Ollama

**Easy local
execution**

Context length;
KV-cache precision

[Documentation ↗](#)

vLLM



**Concurrent-request
serving**

PagedAttention;
continuous
batching;
prefix reuse

[Documentation ↗](#)

SGLang



**High-throughput
serving**

RadixAttention;
prefix reuse;
cache-aware
scheduling

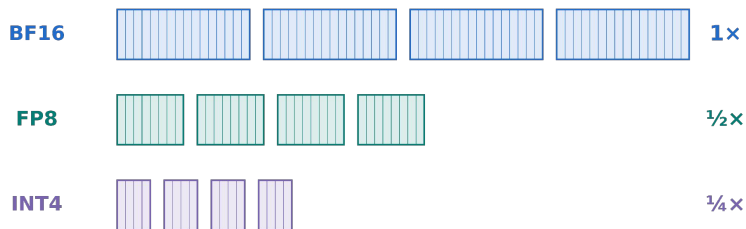
[Documentation ↗](#)

Illustrative strengths, not exclusive features; engine choice depends on the use case.

Can We Store Less? Quantization and Eviction

Reduce bytes per cached element or retain fewer token positions.

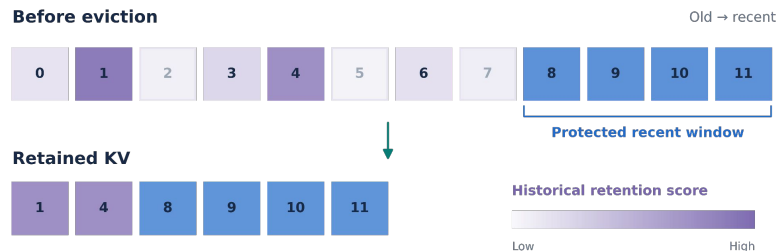
KV QUANTIZATION



Same 4 elements · fewer bits per element

Keep the positions; reduce precision.

KV EVICTION



Illustrative scores · original position IDs preserved

Remove selected KV entries;
discarded states are unavailable.

Both can affect output quality. Ideal payload ratios exclude metadata and higher-precision residuals; speedups vary.

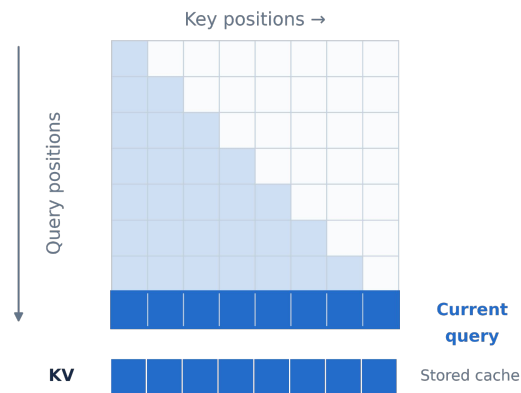
[KVPress ↗](#)

[LMCache ↗ \(offloading & reuse, not eviction\)](#)

Can We Attend to Less? Windowing and Sparsity

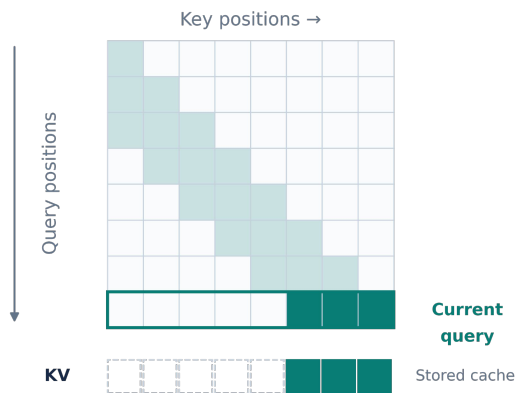
Compare positions stored with positions attended at each decoding step.

FULL CAUSAL



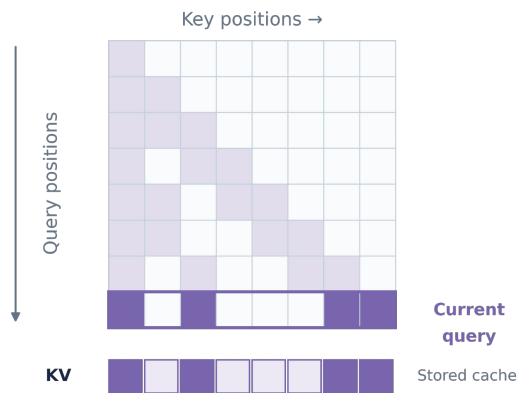
Stored: $O(t)$
Attend: $O(t)$ per step t

LOCAL WINDOW



Stored: $O(W)$
Attend: $O(W)$ per step

DYNAMIC SPARSE



Stored: often $O(t)$
Attend: selection + $O(k)$

Fewer KV reads do not automatically mean less stored KV.

t: context; W: window; k: selected positions. Cache: dark = read, pale = stored, dashed = absent. Selection adds overhead.

Putting Everything Together: Serving Scenarios

Which optimization helps most—and which costs remain?

A Benchmark: independent short prompts

What is 17×24 ?

What is the capital of Peru?

Translate “good morning” into French.

Different inputs · little shared prefix

B Coding assistant: same repository context

IDENTICAL PREFIX

You are a coding assistant. Repository: shop-api.
[README + source files + tests, in fixed order · 8k tokens]

Find the bug
in checkout.

Add tests
for refunds.

Explain the
auth flow.

Each prompt = identical prefix + its new request

Large KV footprint

GQA, KV quantization A B

High attention IO

FlashAttention A B

Memory fragmentation

PagedAttention A B

Many active requests Continuous batching A B

Repeated shared prefixes

Prefix caching B

Long history

Windowing / sparsity B

CAREFUL!!!

What You Need to Remember

Six intuitions to retain; the rest is a toolbox to explore when needed.

1 Inference bottlenecks

Prefill: often compute-bound.
Small-batch decode: often bandwidth-bound.

2 GQA

Share K/V heads to reduce KV storage and reads.

3 FlashAttention

Exact attention with less intermediate memory traffic.

4 PagedAttention

Allocate growing KV caches in flexible physical blocks.

5 KV quantization

Reduce bits per KV element:
precision-memory trade-off.

6 Sliding-window attention

Bound KV storage in local-attention layers.

Know what exists; learn the details by doing, when your use case requires them.

Lecture roadmap

PART 01 KV cache fundamentals

PART 02 Efficient LLM inference

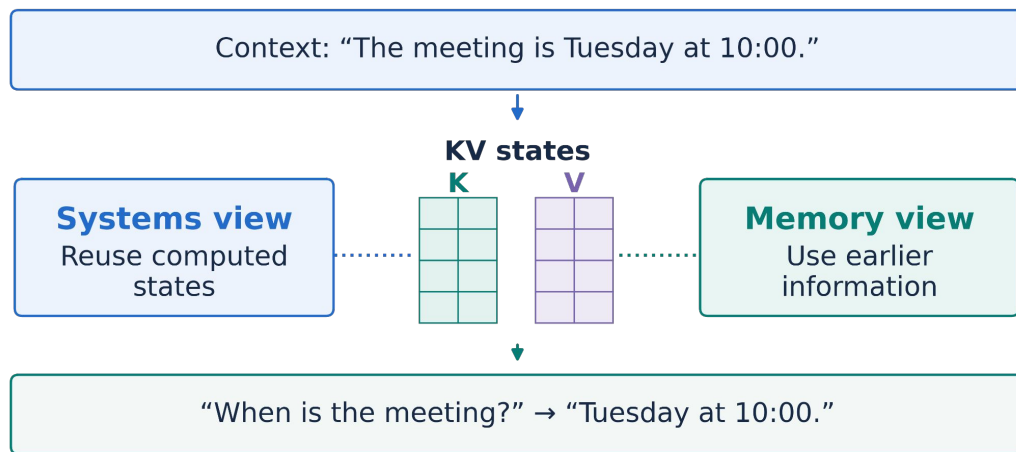
PART 03 Context as memory

PART 04 Advanced topics

PART 05 Key takeaways

From KV Cache to Context as Memory (Intuition)

Two perspectives on the runtime representations of previous context.



PRETRAINED WEIGHTS

Learned during training;
fixed during ordinary inference.

RUNTIME CONTEXT STATE

Representations computed
from the supplied sequence.

EXTERNAL KNOWLEDGE

Selected material brought
into the model's input.

Context supplies information; KV caching makes repeated use cheaper.

In-Context Learning

Same test input. Different demonstrations. No trained weight updates. Connection to Instruction Following.

Same model · fixed pretrained weights θ

PROMPT A

Example 1		→	ALPHA
Example 2		→	ALPHA
Example 3		→	BETA
New input		→	?

Expected: BETA

PROMPT B

Example 1		→	BETA
Example 2		→	BETA
Example 3		→	ALPHA
New input		→	?

Expected: ALPHA

Different context change the prediction; pretrained weights stay fixed.

Attention as Associative Memory

A query scores keys, then combines values with normalized attention weights.

$$\alpha_{ti} = \frac{\exp(q_t^T k_i / \sqrt{d_k})}{\sum_{j \leq t} \exp(q_t^T k_j / \sqrt{d_k})}$$

$$o_t = \sum_{i \leq t} \alpha_{ti} v_i$$



QUERY Retrieval cue

KEYS Match the query

VALUES Content to combine

Example of one head in one layer. Keys/values are learned representations; attention weights are illustrative.

Can Context Act Like Parameter Updates?

An intuition: context can act like changing the effective weights of a small network.

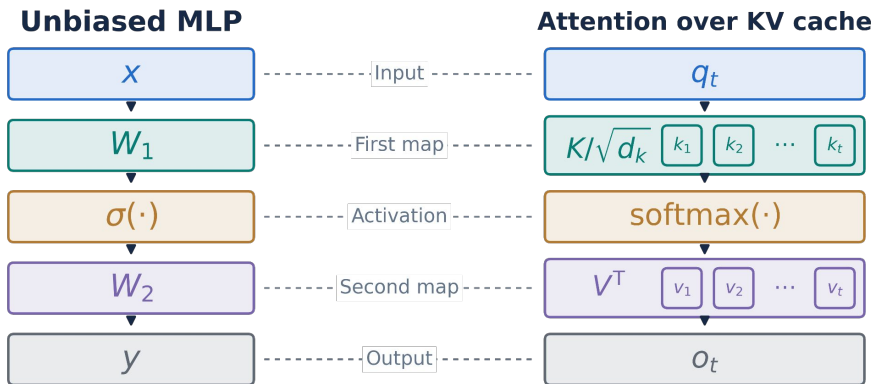
ANALOGY: SAME COMPUTATIONAL ROLES

Unbiased MLP: $y = W_2 \sigma(W_1 x)$

Attention: $o_t = V^T \text{softmax}(K/\sqrt{d_k} q_t)$

$W_1 \leftrightarrow K/\sqrt{d_k}$, $\sigma \leftrightarrow \text{softmax}$, $W_2 \leftrightarrow V^T$

Residual update: $h' = h + \Delta h$.
Simplified view of one attention head.



Analogous roles: MLP activation (e.g. GELU) $\leftrightarrow$ softmax

[Goldwasser et al., ICML 2026](#)

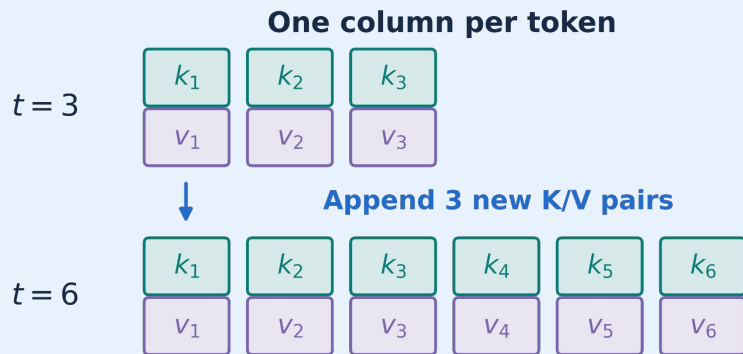
Pretrained weights stay fixed; context changes the network's effective mapping.

Do We Need to Remember Every Token?

Keep per-token K/V, or update a compact state as tokens arrive.

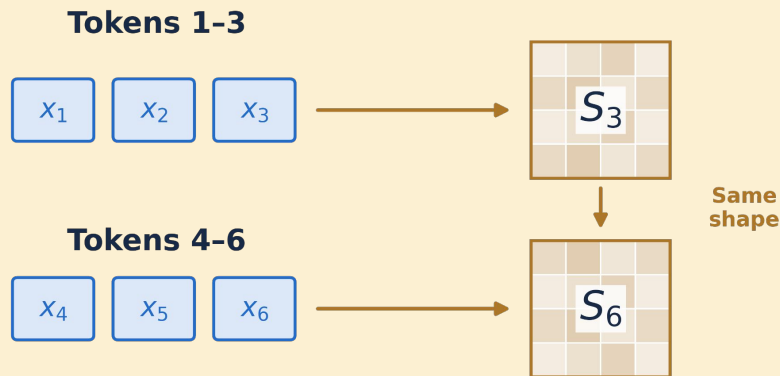
EXPLICIT KV MEMORY

Per-token entries; storage grows with t .



RECURRENT MEMORY

Updated contents; fixed state dimensions.



Fixed-size state $\neq$ lossless compression of full softmax attention.

Linear Attention: A Recurrent Memory

Factorized similarity lets us accumulate history before applying the query.

REARRANGE THE ATTENTION NUMERATOR

$$\sum_{i \leq t} v_i [\phi(k_i)^T \phi(q_t)] = (\sum_{i \leq t} v_i \phi(k_i)^T) \phi(q_t) = S_t \phi(q_t)$$

UPDATE AND NORMALIZED READ

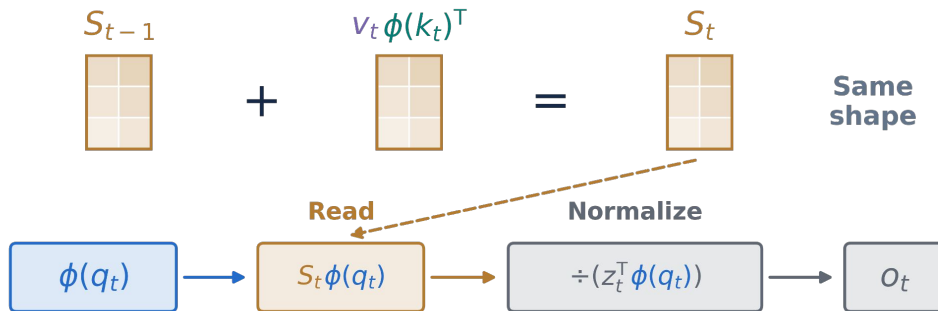
$$S_t = S_{t-1} + v_t \phi(k_t)^T$$

$$O_t = \frac{S_t \phi(q_t)}{z_t^T \phi(q_t)}$$

$$Z_t = Z_{t-1} + \phi(k_t)$$

ϕ : query/key feature map. z_t normalizes the attention weights.

WRITE ASSOCIATIONS, THEN READ

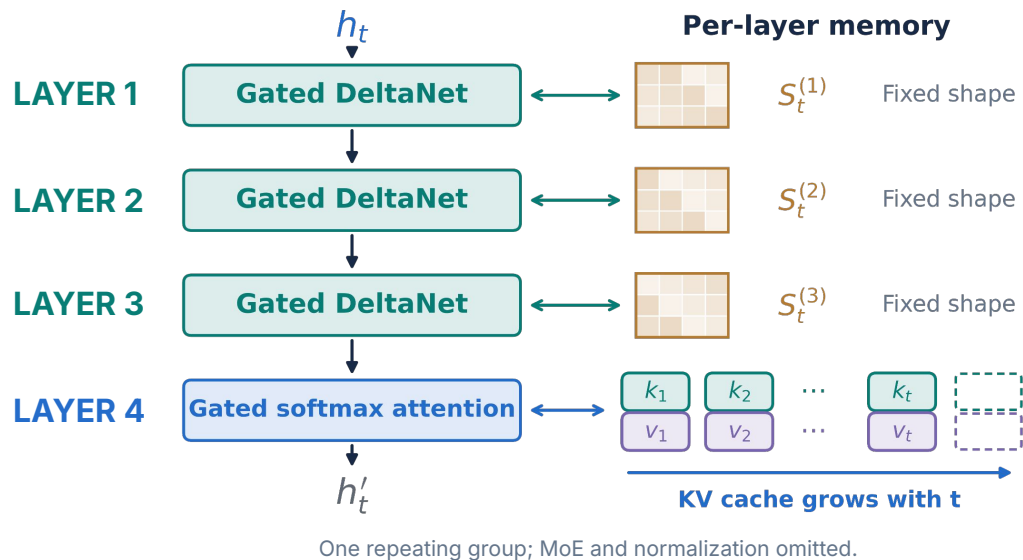


Fixed-size state; a different attention rule from softmax.

[Schlag et al., ICML 2021](#)

Hybrid Architectures: Two Types of Memory

Mix compact recurrent states with attention layers that retain per-token K/V.



GATED DELTANET LAYERS

Updates a compact state; state dimensions stay fixed.

GATED SOFTMAX ATTENTION

Reads per-token K/V; cache grows with context.

EXAMPLE · QWEN3-NEXT

3 GDN layers : 1 attention layer

[Official architecture ↗](#)

Hybrid models retain both compact states and some growing KV caches.

(Optional) From Memory Updates to Test-Time Training (TTT)

Instead of only accumulating associations, learn a key-to-value mapping.

Could the memory state itself be a model that learns from incoming tokens?

A SMALL MODEL AS MEMORY

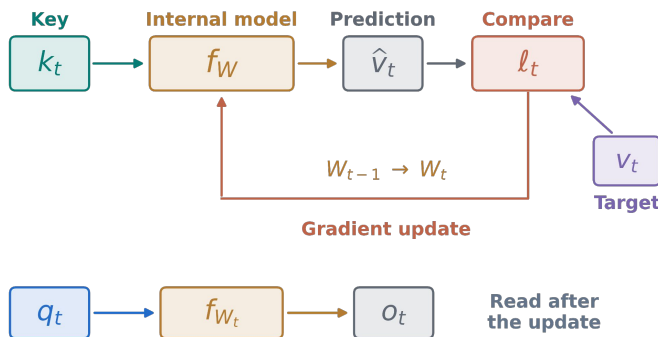
PREDICT → COMPARE → UPDATE

$$\hat{v}_t = f_{W_{t-1}}(k_t)$$

$$\ell_t(W) = \frac{1}{2} \|f_W(k_t) - v_t\|^2$$

$$W_t = W_{t-1} - \eta \nabla \ell_t(W_{t-1})$$

$$o_t = f_{W_t}(q_t)$$



Why learn to predict v_t when we already have it from the projections?

Predicting v_t is the **training exercise** to write into the memory, **not the purpose** of the layer

Update the internal model W ; pretrained outer parameters θ stay fixed.

Illustrative TTT-style layer: k , v and q are learned token views.

[Sun et al., ICML 2025](#)

What Changes During Inference?

Context can change predictions while pretrained parameters remain fixed.

The behavior → In-context learning (ICL): examples in the prompt guide predictions.
Mechanisms below describe how runtime memory is stored and updated.

MECHANISM	UPDATED AT INFERENCE	STATE SIZE VS. CONTEXT
Cached softmax attention	Per-token keys and values	Grows with t
Linear / recurrent layers	Compact state / fast weights	Fixed dimensions
TTT-style layers	Weights of an internal model	Fixed dimensions

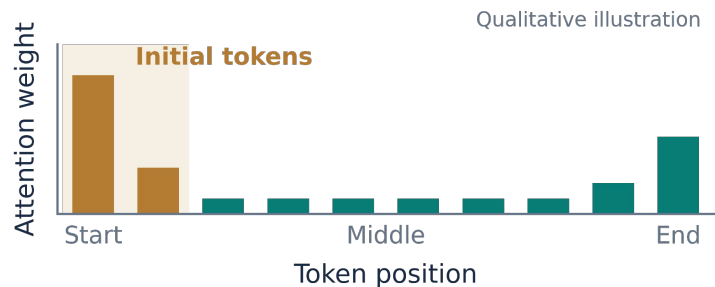
Fixed dimensions assume a fixed architecture; hybrids combine compact states and growing K/V.

Pretrained outer parameters θ stay fixed in all three examples.

Long Context $\neq$ Perfect Memory

Information can remain in context without being used effectively.

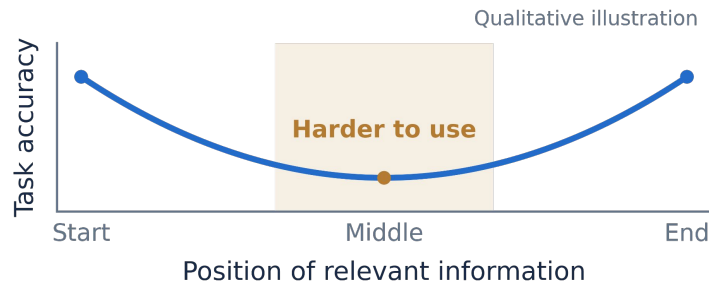
ATTENTION SINKS



Some heads assign high attention to initial, even uninformative, tokens.

[StreamingLLM · Xiao et al. 21](#)

LOST IN THE MIDDLE



Accuracy can fall when relevant information sits in the middle.

[Liu et al. · TACL 2024 21](#)

More stored context does not guarantee effective recall.

What Should We Keep, Forget, or Retrieve?

Long-context memory requires choices about capacity, fidelity and relevance.

KEEP

Retain token-level K/V;
GPU memory grows with context.

COMPRESS

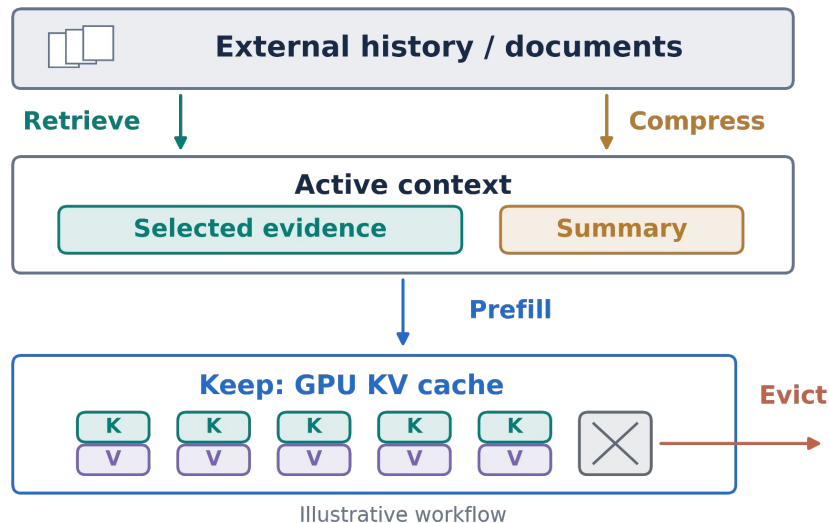
Reduce storage;
some details may be lost.

RETRIEVE

Bring selected external evidence
into the active context.

EVICT

Free active KV space;
predictions may change.



KV Context Management: If context is limited, what should a model remember—and fetch later?

Lecture roadmap

PART 01 KV cache fundamentals

PART 02 Efficient LLM inference

PART 03 Context as memory

PART 04 **Advanced topics**

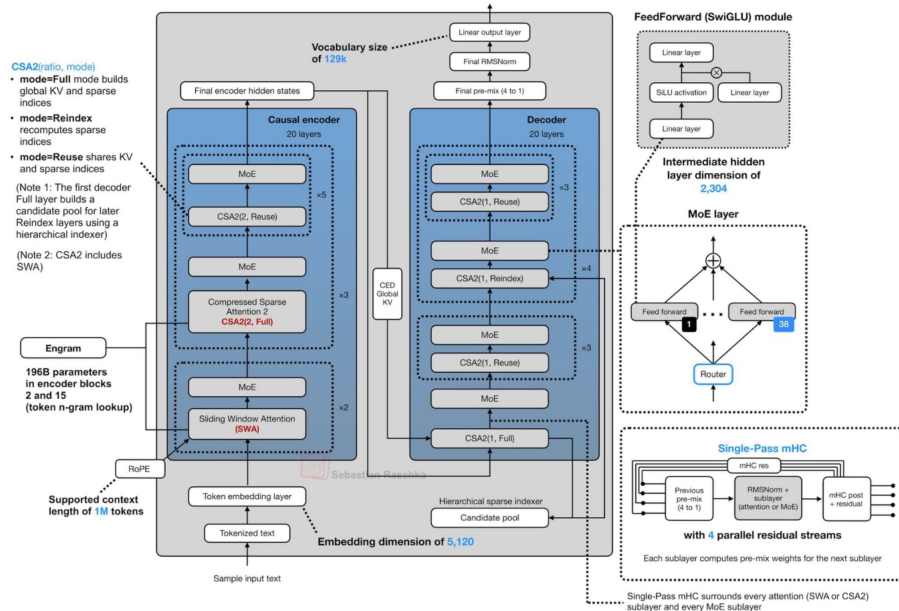
PART 05 Key takeaways

DeepSeek-V4.1-Flash: Integrated Efficiency

Three complementary choices: what to compute, what to store, and what to recompute.

DeepSeek V4.1-Flash (552B)

Text backbone shown. Vision encoder and DSpark speculative decoding modules omitted.



552B MoE backbone

1 COMPUTE LESS DURING PREFILL

Causal encoder-decoder (20 + 20 layers)
8B active prefill / 16B active decode

2 STORE LESS GLOBAL KV

Share global KV across layers + FP4.
890 bytes/token · $\approx \frac{1}{4}$ GPU KV memory

3 RECOMPUTE LOCAL KV

Replay recent tokens to rebuild local K/V.
 $\approx \frac{1}{8}$ persistent cache (host / SSD)

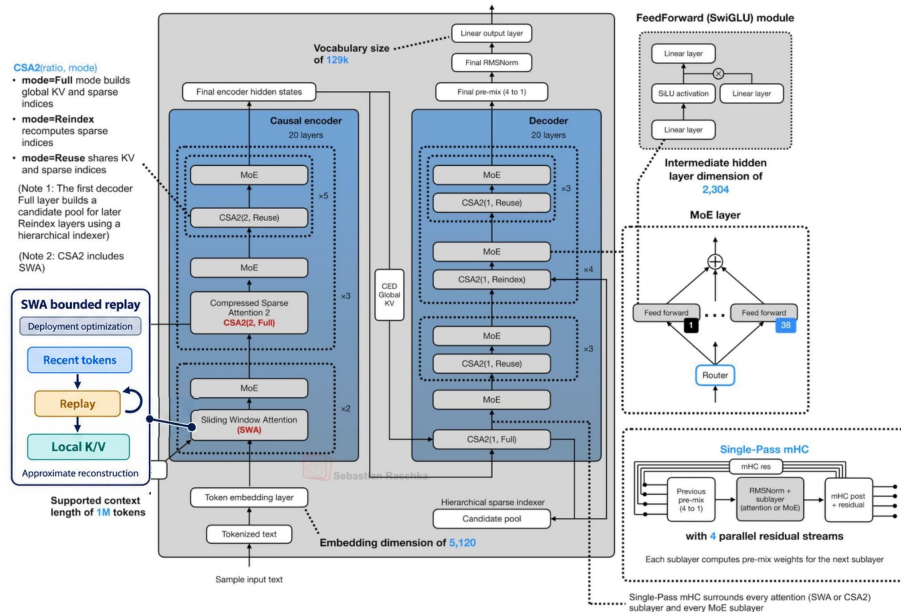
Memory ratios vs. DeepSeek-V4-Flash.
[Technical report](#) · Diagram: Sebastian Raschka

DeepSeek-V4.1-Flash: Integrated Efficiency

Three complementary choices: what to compute, what to store, and what to recompute.

DeepSeek V4.1-Flash (552B)

Text backbone shown. Vision encoder and DSpark speculative decoding modules omitted.



552B MoE backbone

1 COMPUTE LESS DURING PREFILL

Causal encoder-decoder (20 + 20 layers)
8B active prefill / 16B active decode

2 STORE LESS GLOBAL KV

Share global KV across layers + FP4.
890 bytes/token · $\approx \frac{1}{4}$ GPU KV memory

3 RECOMPUTE LOCAL KV

Replay recent tokens to rebuild local K/V.
 $\approx \frac{1}{8}$ persistent cache (host / SSD)

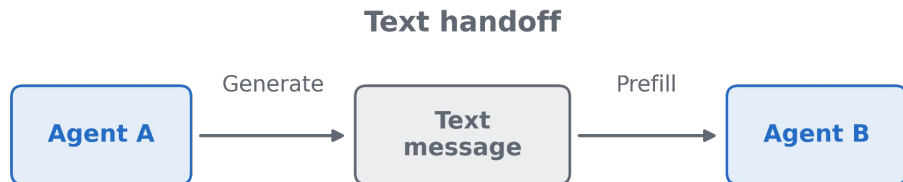
Memory ratios vs. DeepSeek-V4-Flash.
[Technical report](#) · Diagram: Sebastian Raschka

Can Agents Communicate Through KV States?

Research direction: transfer useful KV representations through a learned communication interface.

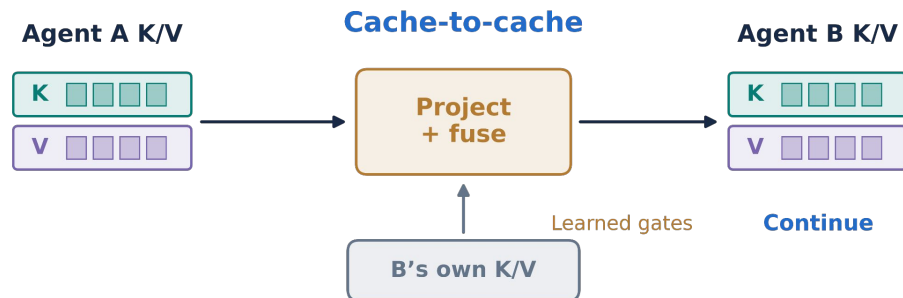
CONVENTIONAL: TEXT HANDOFF

Generate a message token by token.
The receiver prefills that message.



RESEARCH: CACHE-TO-CACHE

Project and fuse source K/V into
selected receiver layers using
learned gates.

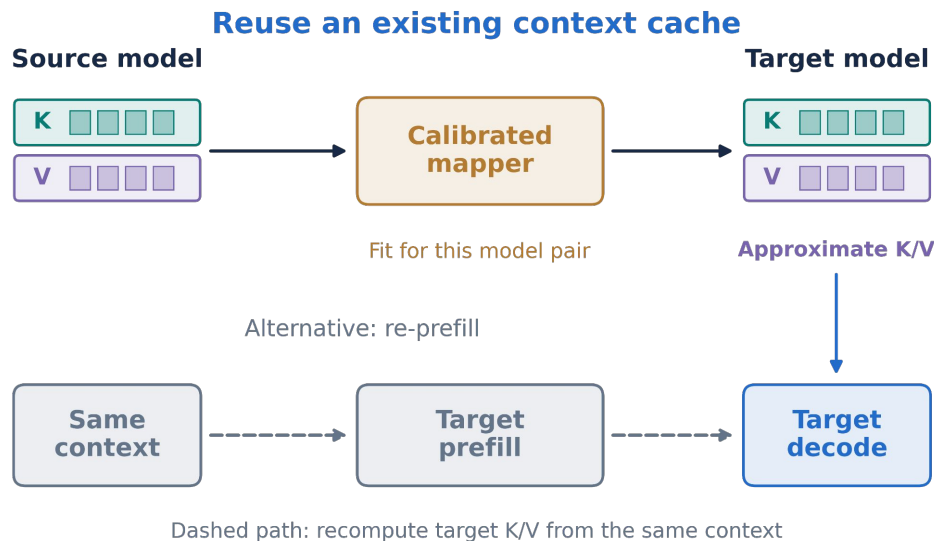


Requires a trained interface; evaluate transfer cost and trust.

[Cache-to-Cache \(ICLR 2026\)](#)

Can We Transfer KV Across Models?

When switching models, can we reuse the context cache instead of prefilling it again?



WHY NOT COPY DIRECTLY?

Same tensor shape $\neq$ same representation.

WHAT RESEARCH TRIES

Calibrate a mapping for a model pair.

THE TRADE-OFF

Save prefill; pay mapping cost.
Validate quality for each model pair.

[Heo et al. \(2026\) · Cross-Model KV Cache Transfer 7](#)

Lecture roadmap

PART 01 KV cache fundamentals

PART 02 Efficient LLM inference

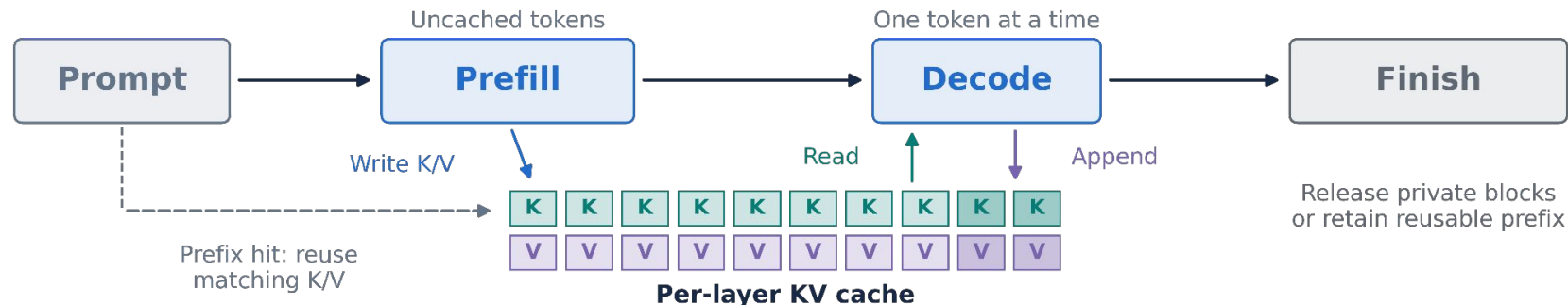
PART 03 Context as memory

PART 04 Advanced topics

PART 05 Key takeaways

The Life of a KV Cache

One request, several optimization layers: choose techniques that address the measured bottleneck.



MODEL

Reduce K/V size

GQA · MLA

KERNEL

Reduce memory I/O

FlashAttention

KV STORAGE

Allocate / compress

Paging · quantization

SERVING

Schedule / reuse

Batching · prefix reuse

Context shapes predictions; caching lets us reuse the computations it produced.

Five Key Takeaways

The essential ideas from efficient inference and context as memory.

01

KV caching avoids redundant computation

Reuse earlier K/V instead of recomputing the entire prefix.

02

Measure prefill and decode separately

Track TTFT, TPOT and throughput; bottlenecks depend on the workload.

03

Efficiency techniques complement each other

Architecture, attention kernels, allocation, scheduling and compression.

04

Context provides useful—but imperfect—memory

Examples and retrieved evidence guide predictions; longer context does not guarantee recall.

05

Know what changes during inference

KV caching stores activations; test-time learning updates an internal model.

Questions?