Lecture 6:
Serving LLMs at Scale

Edouard Oyallon
edouard.oyallon@cnrs.fr
CNRS, ISIR

What you will learn today

- How we serve lots of users at once (and why we obsess over "percentiles" instead of "average")
- Prefill vs decode: reading the prompt vs generating tokens, two phases, very different pain
- How to make waiting feel shorter: "chunked" prefill, streaming + "Thinking" indicators (no magic, just good UX)
- What *test-time scaling* is
- Yes, this is what everyone is fighting over right now (hardware, kernels, serving stacks)

From Training to Inference

- **Training is complete:** we now have a model that can generate an output from a single prompt.

- **Next step: inference & serving:** make the trained model available as a **service/API** that can handle **many requests at once**.

- **Goal:** deliver the *same model behavior* under real-world load, within a Service Level Agreement.

- **Production reality:** models *win or lose* on **fast, stable, predictable serving.** That's what drives user adoption.

- **Training is a sprint, inference is the marathon**: at scale, serving is what keeps the hardware (and costs) running 24/7.

What Makes Inference Production-Ready?

- At inference time, a LLM-service must deliver:
  - **Quality:** accurate, robust, and relevant outputs
  - **Low-latency:** fast responses for a smooth user experience
  - **High throughput:** able to serve many requests in parallel
  - **Controlled cost:** efficient use of compute and infrastructure
  - **Safety & reproducibility:** predictable, auditable behavior *(see previous lecture)*
  - **Reliability & scalability:** stays available and scales with traffic

Inference is Different from Training!

- **Training:** optimize a model once.
- **Inference:** run it millions of times, fast, for real users.
- **Different workload**: training is offline & predictable; inference is online & fragmented
- **Different compute pattern**: training uses forward + backward; inference is forward-only, but generating tokens is sequential, so latency matters
- **Different scaling**: training scales with steps/data; inference scales with concurrent users and request rate
- **Long context dominates cost:** we (re-)process large prompts and maintaining conversation history.
- **User-facing constraints:** many users *want* fast responses; whereas training can be slower and retry

P90, P95 & P99 Latency Metrics

- One needs a metric to measure users waiting time, not only the average case.

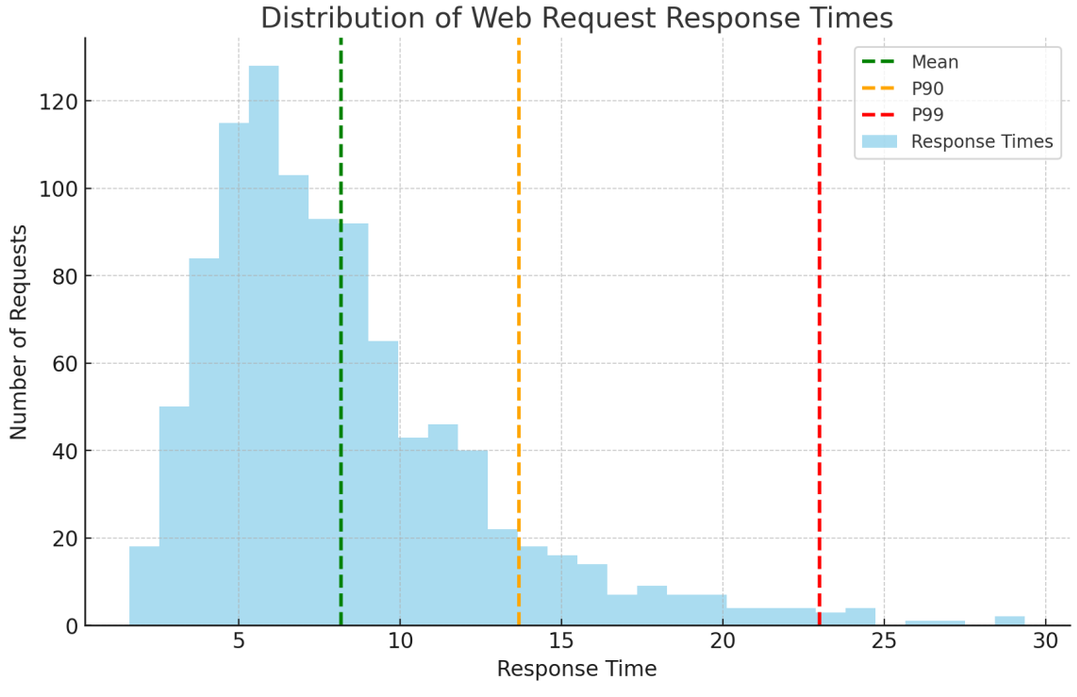
- Instead of just the *average*, we look at **percentiles:**

  - **P90 latency** – 90% of requests are **faster than this value**

  - **P95 latency** – 95% of requests are **faster than this value**

  - **P99 latency** – 99% of requests are **faster than this value**

- It captures the **tail latency**: the slowest requests users actually feel



Performance Metrics

- **TTFT (Time To First Token)**: latency from request start to the first generated token.
- **ITL (Inter-Token Latency)**: time between successive generated tokens (often reported as percentiles).
- Throughput:
  - **Tokens/sec**
  - sometimes split into **prefill tokens/sec** and **decode tokens/sec (next slides)**
- **End-to-end latency**: total time to finish the response (percentiles).
- **Concurrency**: max simultaneous users (or requests/sec) while meeting a latency target.
- **Goodput**: *useful* tokens/sec.
- **Memory footprint**: peak GPU memory + impacts max context + max concurrency.
- **Error rate**: Out of Memories, timeouts, under load.

Generating Tokens

Probabilistic view

- One inference step:
  - Input tokens → embeddings
  - Forward Pass through transformer layers
  - **Output:** logits for the next token (a score for each vocab item)
  - Choose next token according to decoding rule
  - **Append token and repeat** until stop token or max length.
- Probabilistic framing:
  - LLM first inputs an user request $(x_1, ..., x_n)$.
  - Inference: repeatedly computing

$$x_{n+t+1} \sim p\left(\cdot \mid \underbrace{x_1, \ldots, x_n}_{\text{user / prompt}}, \underbrace{x_{n+1}, \ldots, x_{n+t}}_{\text{generated so far}} \right)$$

    and choosing a token at each step.

Decoding Strategies

- Given a context $x_1 \dots x_{t-1}$ the model outputs logits z_t for a temperature β and set

$$p_\beta(x_t \mid x_{<t}) = \text{softmax}\left(\frac{z_t}{\beta}\right)$$

- **Greedy decoding (argmax):** choose the most likely token each step

$$x_t = \arg\max_{y \in \mathcal{Y}} \; p_\beta(y \mid x_{<t})$$

  + **Fast, deterministic**
  − Can be **repetitive** and may get stuck in low-diversity loops

- Sampling: draw $x_t \sim p_\beta(. \mid x_{<t})$

  + **More diverse**, often better for open-ended generation
  − Less predictable; may reduce factuality if too random

- **Top-k** sampling: restrict to the k highest-probability tokens, renormalize, then sample.
  + Limits extremely unlikely tokens
  − Quality depends on k ; too large ≈ plain sampling, too small ≈ near-greedy

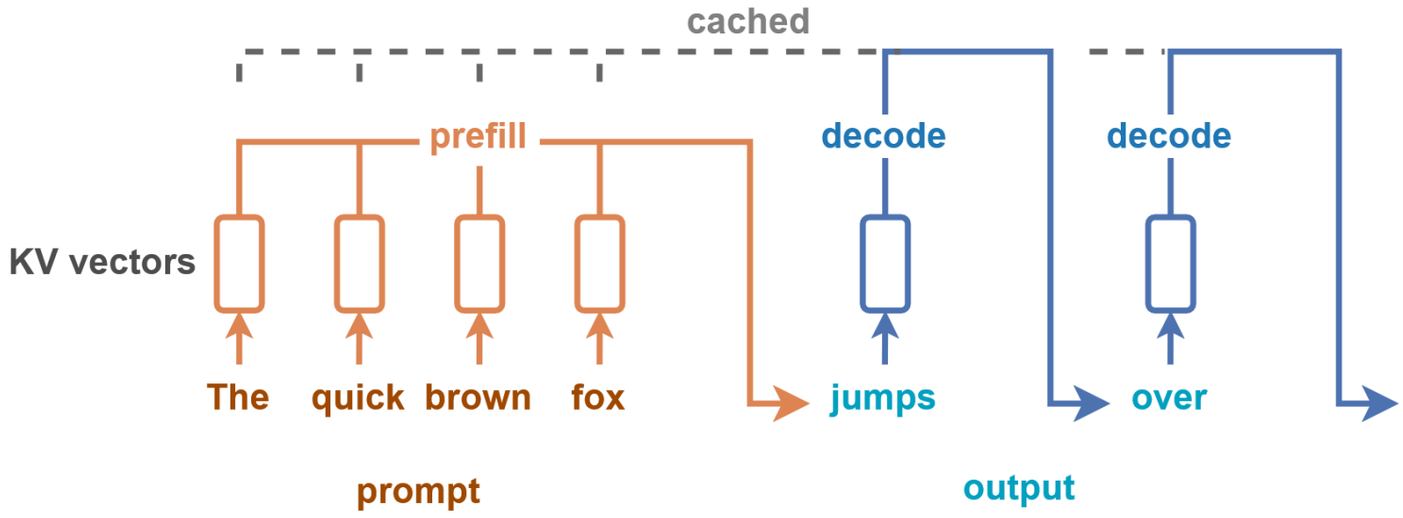
Outline

- Course roadmap:
  - **Token generation basics:** prefill vs decode, KV cache, and what drives TTFT vs ITL
  - **LLM serving:** batching, scheduling policies, and streaming behavior under load
  - **Inference optimizations:** quantization, caching, test-time scaling...
  - **Inference stack (end-to-end):** model → runtime (e.g., vLLM)

Prefill and Decode

The Two Inference Phases: Prefill and Decode

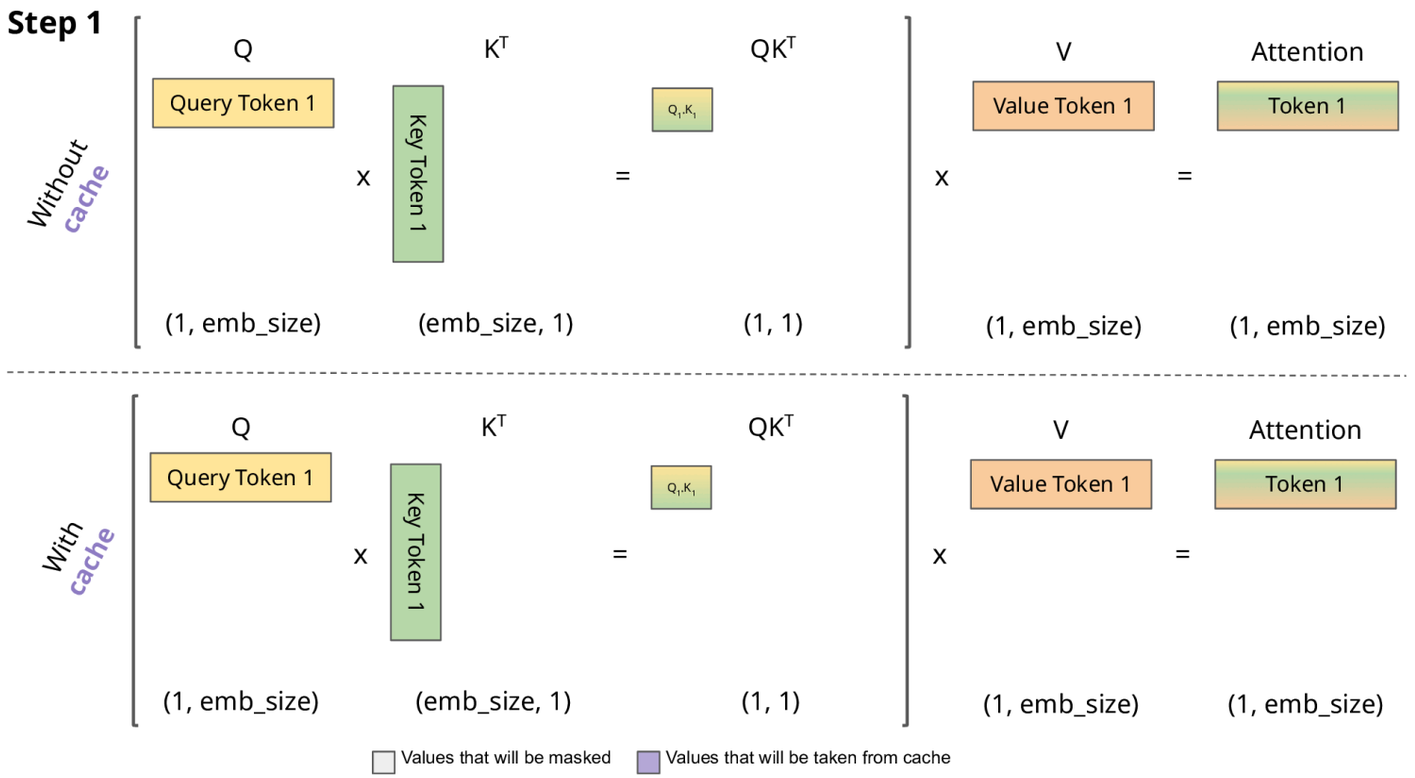
- The Two Inference Phases: Prefill and Decode
  - **Efficient generation relies on a Key-Value (KV) cache**: we store attention keys/values from previous tokens so we don't redo work.
  - For the next few slides, assume a single request (ignore concurrency).
- When serving one request, inference splits into two phases:
  - **Prefill (prompt processing):**
    Process the *entire* prompt/history **once**, compute activations needed for attention, and **initialize the KV cache** for all prompt tokens.
  - **Decode (token generation):**
    Generate **one token at a time**. Each step uses the **existing KV cache** (so we don't recompute the full prompt), then **appends** the new token's K/V entries to the cache.



What is a KV-cache?

- Autoregressive decoding generates one token at a time; naïve attention would repeatedly process the entire prefix, which gets more expensive as the context grows.

- **KV cache** stores each layer's **Keys and Values for past tokens** so we don't recompute them every step; at step t we compute Q for the new token and attend to cached K/V. *Inference builds this KV-cache.*



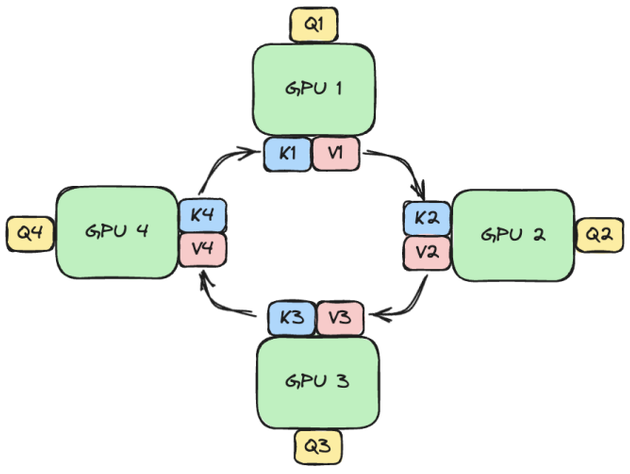
- This speeds up **decode**, but the cache **grows linearly** with sequence length and layers → memory/bandwidth become the bottleneck for long-context. Corresponding footprint: $2 \cdot n_{\text{kv}} \cdot d_{\text{head}}$

Context parallelism: Ring attention

- **Motivation:** very long contexts make **attention activations** + **KV cache** too large to fit on a single GPU. It reduces the memory use BUT increases communications

- Core idea (sequence sharding): split the sequence across N GPUs so each GPU owns a chunk of tokens (its local Q or/and KV blocks)

- **Ring Attention pattern:** each GPU **keeps its Q (resp. KV)** and **rotates KV (resp. Q) blocks** around a ring; it accumulates partial attention results as new KV blocks arrive



$$\sigma(qk^T)v$$

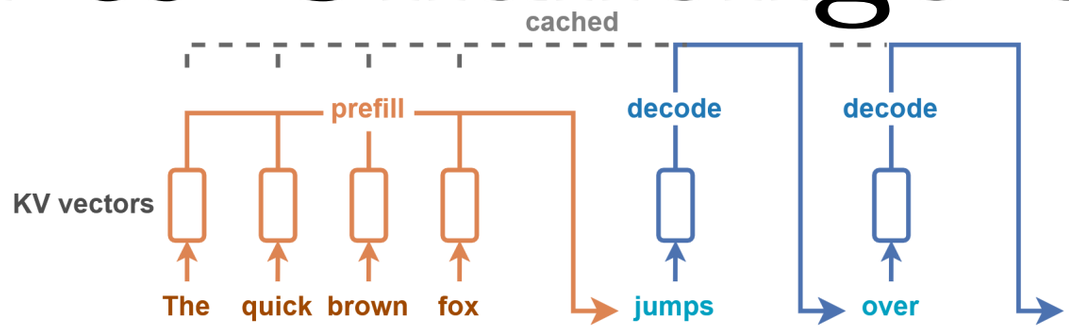
- **Memory effect:** per-GPU memory scales roughly like $\frac{1}{N}$ for sharded KV cache.

- **Communication cost:** adds **traffic** (Q/KV exchange); if compute per step is small (decode / small Q), comms can land on the critical path

- **Key optimization: overlap communication with compute** (blockwise attention) so KV/Q transfer happens while computing attention on the current block

Prefill

Goal & Challenge of Prefill



- **Prefill = prompt pass:** run the model over *all prompt tokens* to (1) produce the initial next-token logits and (2) **materialize the KV cache** (per layer, per token).

- **Objective:** build the KV cache **fast and memory-efficiently** so decode can reuse it (no prompt recomputation) and **time elapsed stays low**, even for long prompts. ***The key metric is TTFT.***

- **Compute scaling**: longer context ⇒ more attention work ⇒ higher time-to-first-token

- **Memory**: KV cache allocations stress memory traffic and footprint.

- **Variable prompt lengths**: padding causes wasted compute/memory, and dynamic KV allocation can cause memory fragmentation / higher tail latency.

Prefill Context Parallelism

- To maintaining high throughput, 3D parallelism (DDP, TP, PP) is used. In addition, there are two strategies (choose based on memory budget) for using Context Parallelism:

- Partial Q, full KV (*accelerate prefill*):

  - each GPU handles a chunk of Q tokens, while KV are replicated so it can attend over the *full* prompt, best when the prompt is *moderately long* (full KV fits).

  - **Tradeoff**: can speed up prefill & reduce TTFT, but adds communication and replicated KV memory.

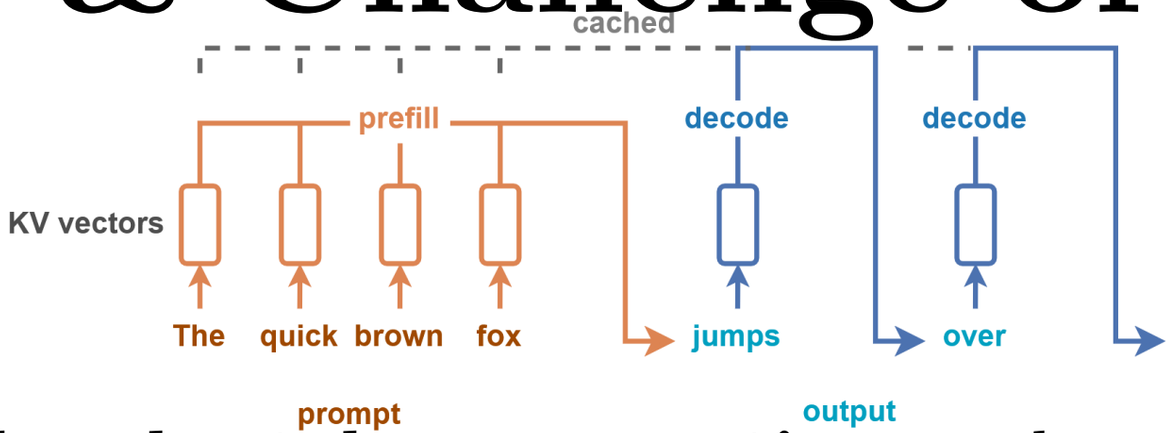
- Partial Q, partial KV (*fit very long prompts*)

  - each GPU keeps only its **local Q, KV chunk** and computes attention by **exchanging KV chunks** (ring style), best when the prompt is *too long* to fit.

  - **Tradeoff:** per-GPU KV memory drops to about $\frac{1}{N}$ of the sequence-dependent state, but **communication increases** and must be **overlapped with compute** to avoid latency penalties.

Decode

Goal & Challenge of Decode



- **Decode = token-by-token generation**: each step depends on the previous token → **sequential over time (limited parallelism)**

- **Uses the KV cache**: at every step, attend over cached KV for all prior tokens + write new KV for the new token

- **Objective**: maximize **tokens/sec** while keeping **latency per token** stable as sequences grow. ***The key metric is ITL.***

- **Memory-bandwidth bound**: each step **reads lots of KV** (grows with context length) → memory traffic dominates

- **Growing working set**: longer generations → larger KV → higher per-token cost

Decode Context Parallelism

- To maintaining high throughput, 3D parallelism (DDP, TP, PP) is used.
- **Autoregressive constraint:** within a single sequence, tokens must be generated **one after another** → *no parallelism across time steps.*
- **The decode bottleneck:** each step uses **tiny Q** (often 1 token) but reads **large KV** (all prior tokens) → memory + communication become critical.
- **Decode Context Parallelism:** shard the **KV cache across GPUs** so each GPU stores only a fraction of the past context; reduces per-GPU memory and increases concurrency, but requires **cross-GPU aggregation** during attention.
- Core tradeoff: More sharding ⇒ less KV per GPU but more communication per token (can hurt latency if not overlapped).

Beyond Single-token Decoding

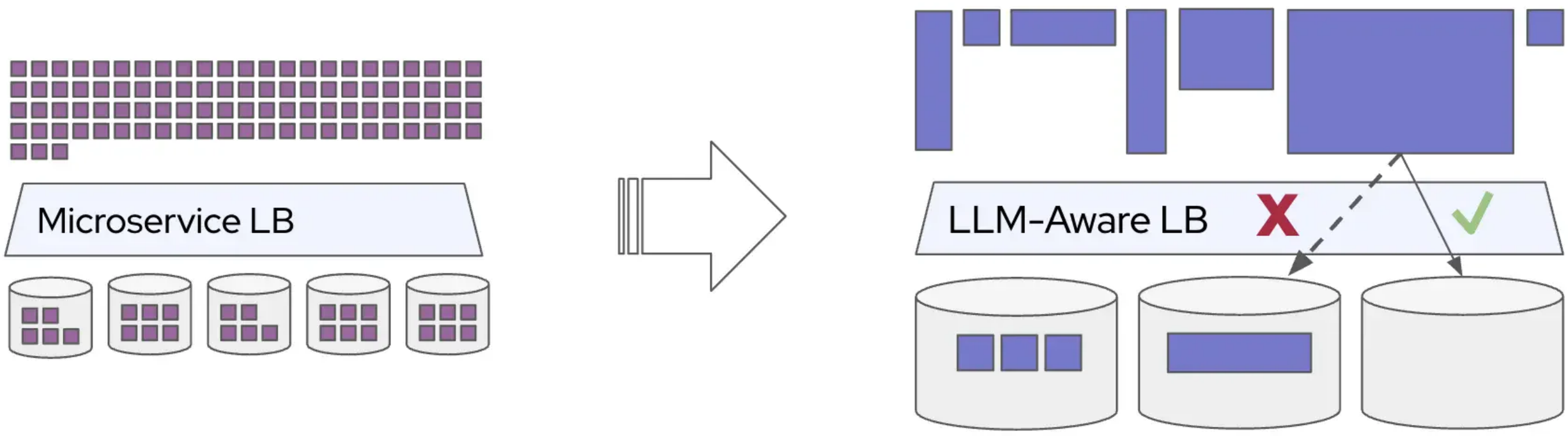
- **Bottleneck:** autoregressive decoding is **sequential** → **1 token per step** → high latency for long outputs
- **Idea:** increase **tokens-per-step** while preserving quality
- Speculative decoding:
  - Small draft model proposes a span of tokens
  - Large model **verifies in parallel** → accept many tokens at once when agreement is high
- Medusa (multi-token heads):
  - Attach lightweight heads to predict **several next tokens simultaneously**
  - Main model validates/chooses consistent continuation → fewer decode iterations
- Diffusion-Transformers (specifically, Masked Diffusion Model):

$$x_t = \mathrm{Mask}(x_{t-1};\ \gamma_t) \quad \text{with} \quad \mathcal{L}(\theta) = \mathbb{E}_{t,x_0}\Big[-\gamma_t \log p_\theta(x_{t-1} \mid x_t)\Big]$$

  - Generate via **parallel refinement steps** (not strict left-to-right)
  - More parallelism potential, with **quality/latency tradeoffs**
- **Key tradeoff**: speedup depends on acceptance rate / verification cost / extra heads / quality

Serving Millions of Users

Difference between LLM server and HTTP server



- **HTTP server:** short, mostly stateless request/response with CPU/I/O + simple load balancing;

- **LLM server:** long-running, often **streaming** responses with big prompts/context, GPU-bound with **special scheduling**

Challenges of serving LLMs at scale (millions of users)

- **Serve** = expose the model via a **stable API/endpoint** that accepts requests and returns outputs reliably.

- **Concurrency** = how many sequences are "in flight" at once on a GPU.

- **Service Level Objective under load:** run inference on demand for many users while meeting **p95/p99 latency** and **uptime** targets.

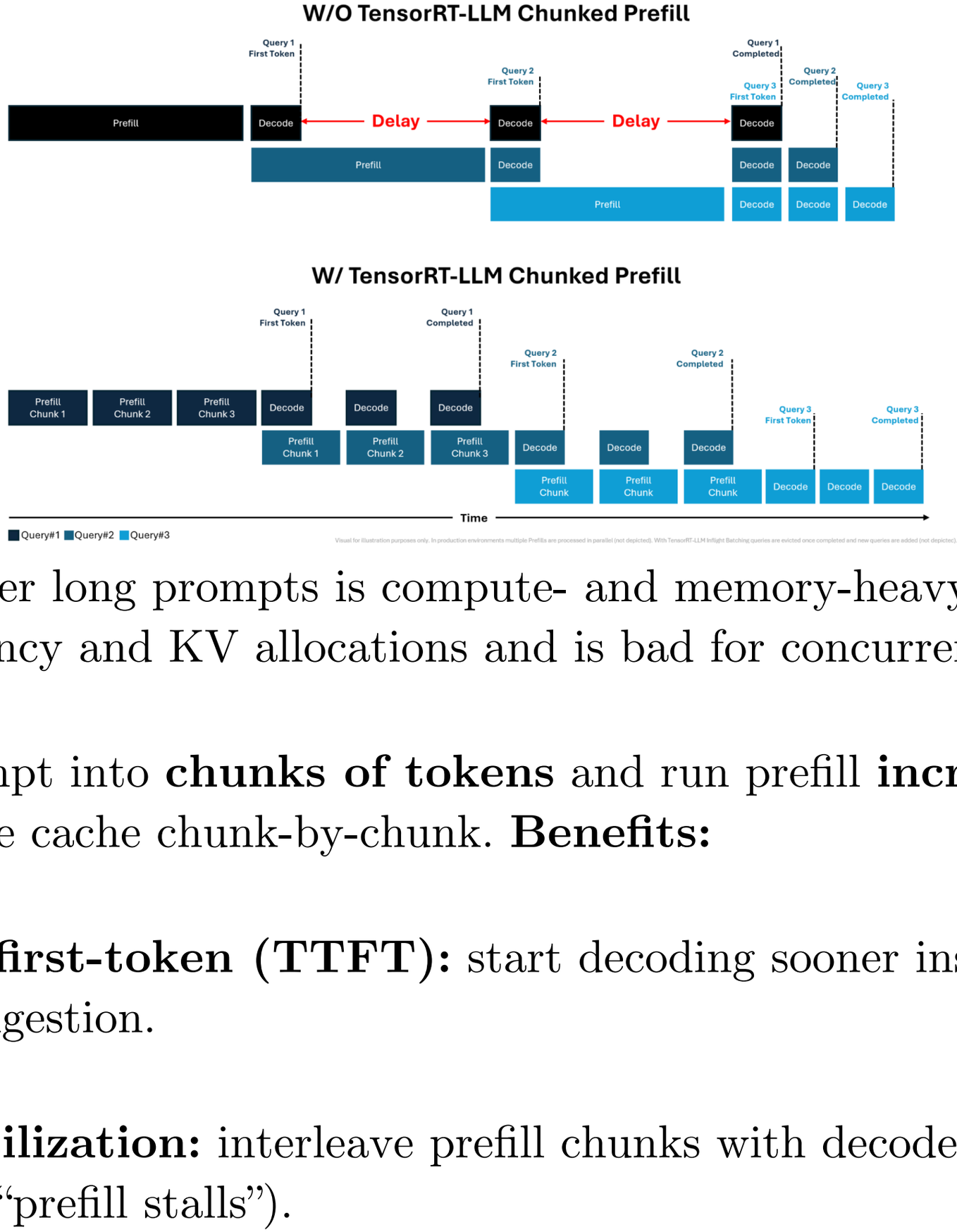
- **Memory-limited inference:** GPU RAM is the main bottleneck (weights + **KV-cache** that grows with context length), which caps batch size and concurrency.

- **Concurrency & scheduling:** prefill vs decode compete for resources; you must decide **who decodes when** and control queueing.

- **Traffic bursts & reliability:** demand is spiky → need admission control, load balancing, and graceful degradation under overload.

- Unlike typical services, LLM "jobs" have variable size: both prompt length and generation length vary widely across requests.

The necessity to "Chunk" Prefill



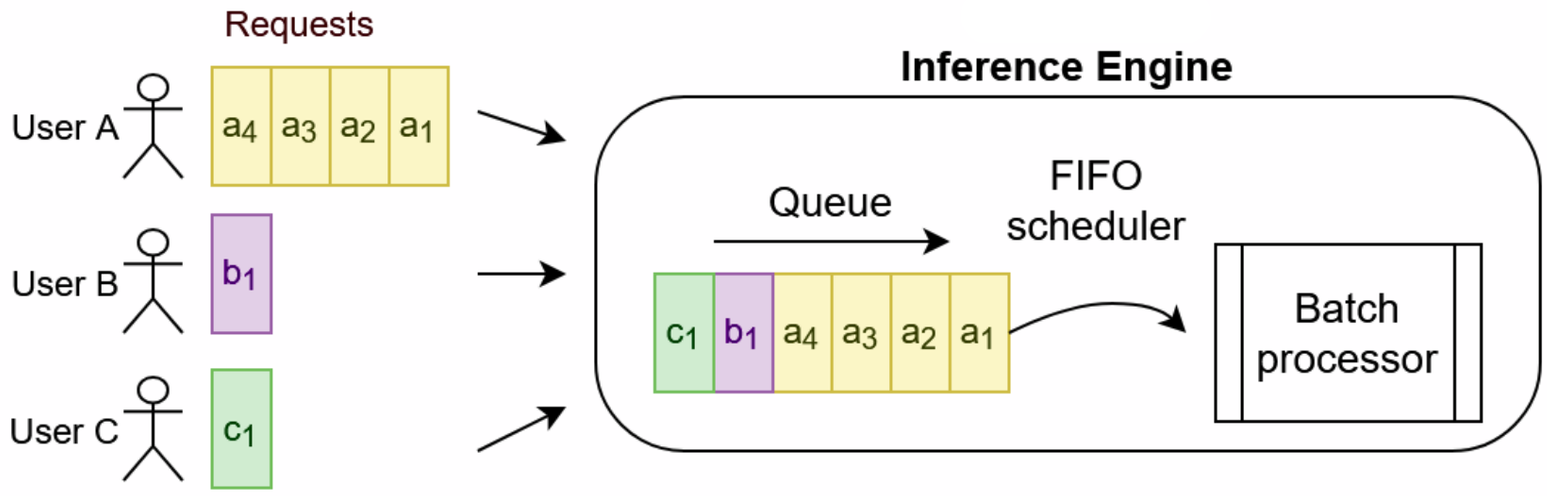
- **Problem:** Prefill over long prompts is compute- and memory-heavy; a single huge prefill can spike latency and KV allocations and is bad for concurrency.

- **Idea:** Split the prompt into **chunks of tokens** and run prefill **incrementally**, appending KV to the cache chunk-by-chunk. **Benefits:**

  - **Lower time-to-first-token (TTFT):** start decoding sooner instead of waiting for full prompt ingestion.

  - **Better GPU utilization:** interleave prefill chunks with decode work (smoother scheduling / less "prefill stalls").

Scheduling chooses which Token-work runs Next

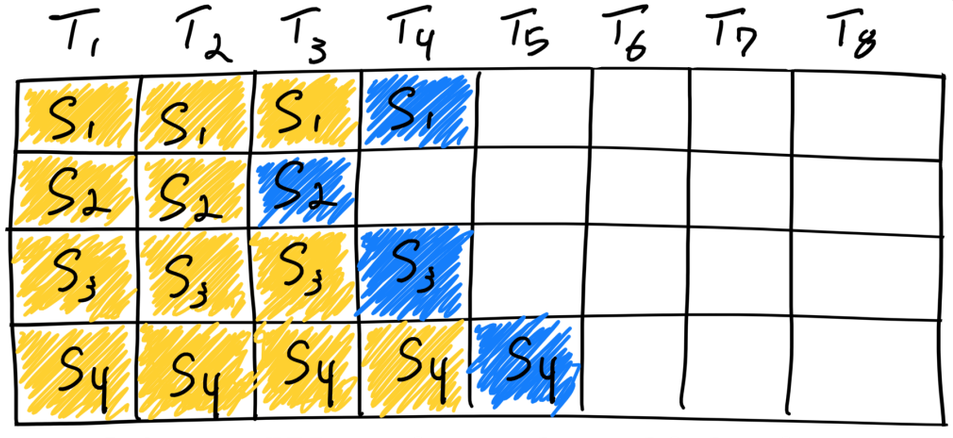


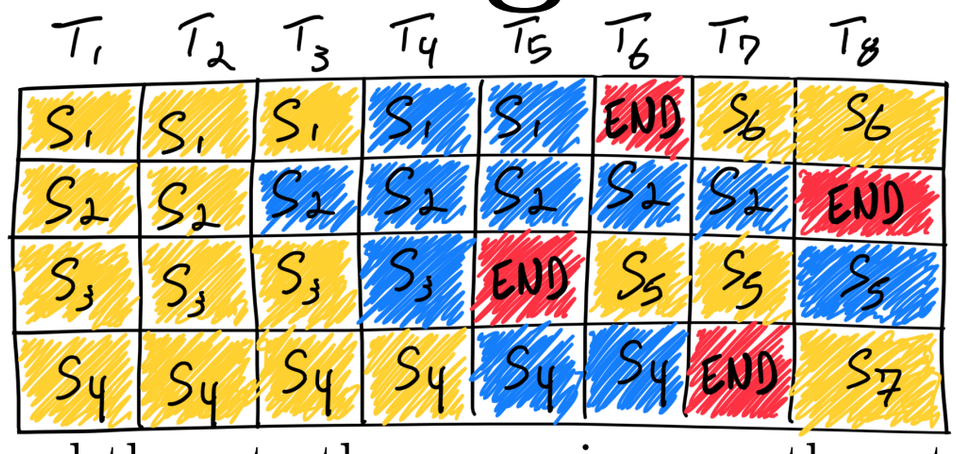
- **Natural idea:** split each request into **schedulable units** (small chunks of work):
  - **Prefill units:** chunks of prompt tokens (micro-batches)
  - **Decode units:** one (or a few) decode steps per active sequence (micro-batches)
- **Scheduler goal**: maximize throughput while meeting latency service constraints, under GPU compute and KV-memory budgets.
- Each request is tracked as a state machine:

  queued $\rightarrow$ **prefill** $\rightarrow$ **decode** $\rightarrow$ **finished** (or aborted)

- Typical queue structure:
  - **Prefill queue:** new prompts waiting to be processed
  - **Decode queue:** active sequences waiting for their **next decode step**

Batching Strategies





- **Static batching:** Wait for a handful of requests, pad them to the same size, run them together.
  Fast and simple, but you either wait to form the batch or waste compute on padding.

- **Selective batching:** Batch only requests that "look alike" (typically group all not non-attention operations together).
  Less padding, fewer slow requests dragging fast ones down.

- **Ragged batching:** Run variable-length sequences together **without padding** (pack tokens + offsets), so work scales with real tokens not max length.
  Less wasted compute and memory, but it needs more complex kernels.

- **Dynamic batching:** Keep batching as requests arrive (small time window), instead of forming one fixed batch.
  Better throughput when traffic is uneven.

- **Continuous batching:** Batch at the *token* level during generation: every step, run all active sequences, drop finished ones, add new ones.
  Keeps the GPU busy even when requests start and finish at different times.

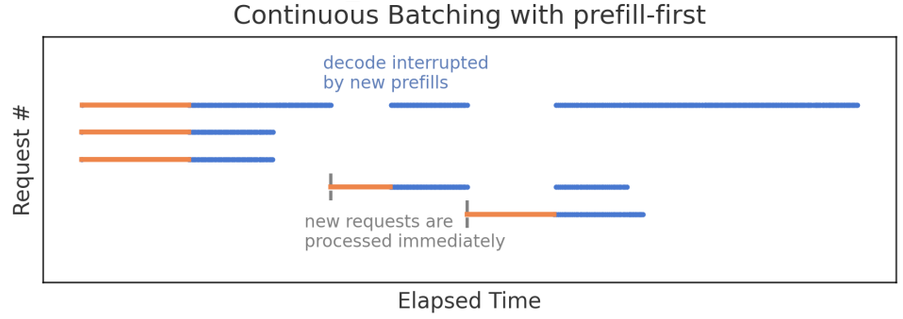
Serving Policies

- **Core challenge:** due to GPU sharing, one "heavy" phase can increase queueing delay, latency.

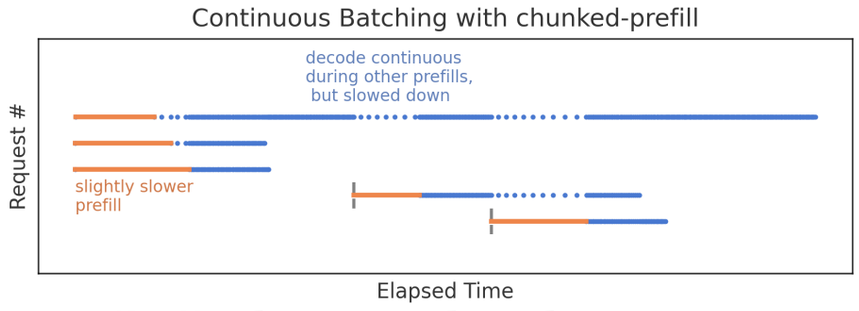
- **FIFO (First-in-First-Out):** arrival order feels fair, but **a long prefill/generation delays other users. Thus, there are** unstable **time-to-first-token (TTFT)** under mixed workloads. **Priority classes (paid vs free users):** improves latency for privileged traffic, yet it obviously starves low-priority traffic.

- Two strategies (among many which have similar tradeoffs):

  - **Policy Prefill-first:** prioritize running **prefills** for new requests to start responses quickly; downside is prefills can monopolize compute and **interrupt decode**: inter-token latency (ITL) could spike but TTFT is good.



  - **Policy chunked prefill:** prioritize decode during each cycle to improve ITL, then use remaining **token budget** for **chunked prefills.**



- **Chunk size** is tunable: smaller chunks protect streaming ITL, larger chunks improve TTFT.

- **A solution is disaggregated serving:** split **prefill (compute-heavy)** and **decode (memory-heavy)** onto different resources to improve both TTFT and ITL.

"Thinking. . . " Indicators: Managing Wait Time in LLM UIs

- Even with better serving, users still experience **wait time** (TTFT, ITL)

- **An indicator** a lightweight **status/progress cue** (e.g., "Thinking. . . ", spinner, step label, partial streaming text) shown while generation is in progress

- It improves **perceived responsiveness** and reduces abandonment when delays are unavoidable (classic progress-indicator effect) *but* it should not fake precision or jumpy updates; mismatch between indicator and actual progress hurts trust.

- My personal assumption: there are two streams

  - **Hidden reasoning stream:** internal scratchpad tokens (not shown, for safety reasons)

  - **Visible answer stream:** the final response tokens that are streamed from it to the user, via an ad-hoc tiny model.

- **Design implication:** keep the UI responsive with a clear "working" state, but only stream what you're willing to show as the user-facing answer

Reducing Memory Pressure

Agressive Quantization

- **Goal:** cut memory + bandwidth to improve throughput, p95 latency, and long-context concurrency.

- **Post-Training Quantization:** weights **W8A16 → W4A16**; optionally quantize **KV cache (KV8/KV4)** to scale long contexts

- **Make INT4 viable:** use stronger granularity (**per-channel**/**per-row**, often **group-wise 32–128**) instead of per-variable

- **Calibrate, don't guess:** set scales/clipping on representative prompts (activation stats) to reduce quantization error

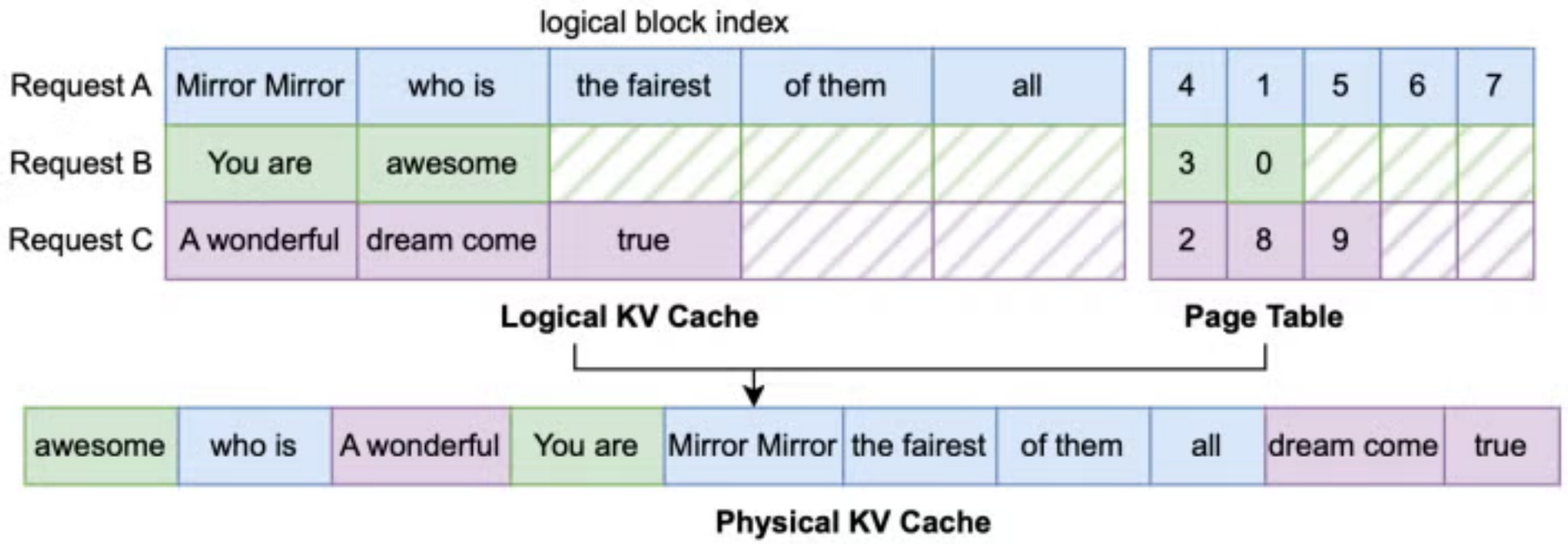
- **Stabilize accuracy:** handle outliers + keep sensitive parts in higher precision (e.g., embeddings/LM head/outlier channels in FP16); apply smoothing/ rescaling (GPTQ-style, AWQ-style, SmoothQuant-like)

- **Reality check in production:** benefits require optimized INT4/INT8 kernel; keep a **fallback** (less-quantized) model for hard queries

KV-Cache Mitigation

- **Prompt caching**: reuse KV for repeated system prompts across requests (big win for chat-style workloads!)

- **Paged attention**: store KV in fixed-size blocks to reduce GPU memory fragmentation and enable efficient allocation/reuse (*used by vLLM!*)



- Architectural variants reduce cache cost: **Multi-Query Attention,** cross-layer sharing, and **compressed KV (Multi-head Latent Attention)**.

$$h_t = D\,x_t \in \mathbb{R}^r, k_t = U_K\,h_t, v_t = U_V\,h_t$$

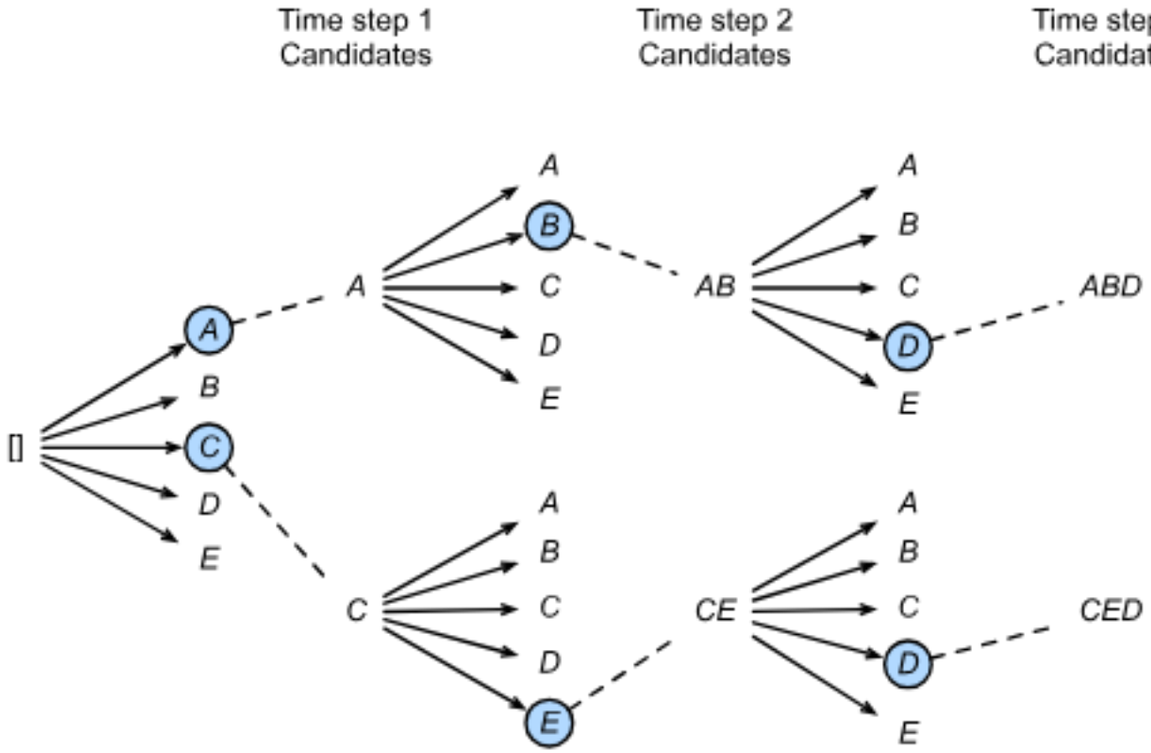
$q_t = Qx_t$ where r is much smaller than the ambient dimension

Boosting Inference

Test-Time Scaling: More Compute at Inference

- A few vocabulary:
  - **Test-time scaling:** allocate more inference resources (e.g., more GPUs) while generating an output.
  - **Test-time compute:** total inference work used per request, often measured in **FLOPs** (and reflected in latency/cost).
  - **Test-time search:** extra exploration at inference to find a better answer (e.g., multiple candidates, verification).
- **How you "spend" it:** longer generation, larger sampling, tool calls, self-checks, multi-step reasoning.
- **Tradeoffs:** improved quality vs increased latency, cost, and sometimes diminishing returns.

Beam Search



- **Beam Search is** a decoding algorithm that keeps the **top** B**-partial sequences** (the "beam") while generating tokens.

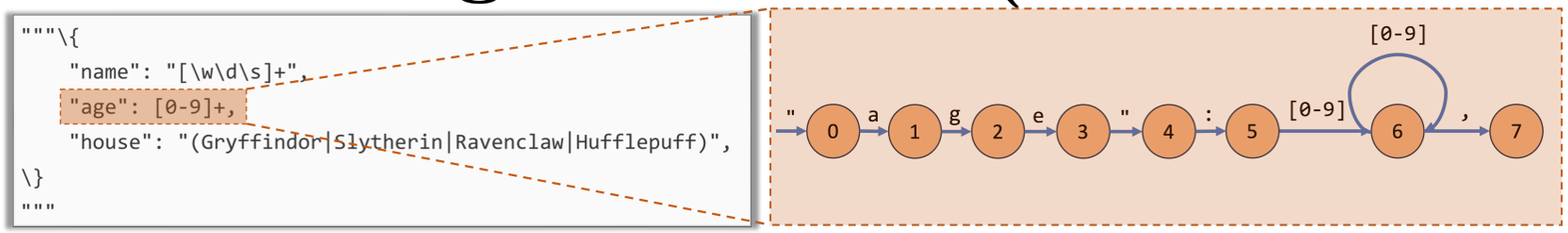
- **How it works:** at each step, **expand** each beam with candidate next tokens → **score** (sum of log-probs) → keep the best B → continue exploration.

- **Scoring:** maximize sequence likelihood, often with **length normalization** to avoid favoring short/long outputs.

- **Why use it:** more reliable than greedy decoding; reduces obvious local mistakes and improves consistency.

- **Tradeoffs:** higher compute and can reduce diversity compared to sampling.

Guided Decoding with FSM (Finite State Machine)



- **Goal:** force the LLM output to follow a **grammar** / **schema** / **allowed format** (e.g., JSON, tool-call syntax)
- **Idea:** represent constraints as an **FSM**. It encodes "what prefixes are valid so far."
- At each decode step:
  - Compute the model logits as usual
  - Query the FSM for the set of allowed next tokens from the current state
  - **Mask** all disallowed tokens (set logits to $-\infty$), then sample/argmax from the remaining ones
  - **State update:** after choosing token, advance FSM
- **Benefits:** guarantees **well-formed outputs**, fewer parsing failures, safer tool calls, less post-processing / retries
- **Tradeoffs:** can reduce diversity, and depends on tokenizer alignment (grammar must be expressed in tokens)

Boosting Reasoning

YaRN: Efficient Context Window Extension

Ref.: YaRN: Efficient Context Window Extension of Large Language Models, Peng et al.

- **Goal:** extend an LLM's usable **context window** beyond its pretraining length (for RoPE-based models).

- **Core idea:** modify RoPE so the model can extrapolate to longer sequences without retraining from scratch.

- **Stabilization trick:** applies an **attention scaling** t adjustment to keep attention behavior well-behaved at longer lengths

$$\text{softmax}\left(\frac{Q^\top K}{t\sqrt{d}}\right)$$

- **How you use it:** pick a t, and then fine-tune briefly on long-context sequences

- **Why it's "efficient":** with these changes, YARN reports reaching strong long-context performance with **far fewer training tokens/ steps** than prior extension recipe (10x)

...leading to Longer Chain of thought

- **Chain-of-thought (CoT):** prompting/decoding style where the model reasons through intermediate steps before the final answer.

- **It enables** better performance on multi-step tasks (math, logic, planning).

- **How it's triggered:** "think step by step", few-shot examples, structured prompts, or models trained to reason, typically via prompts like:

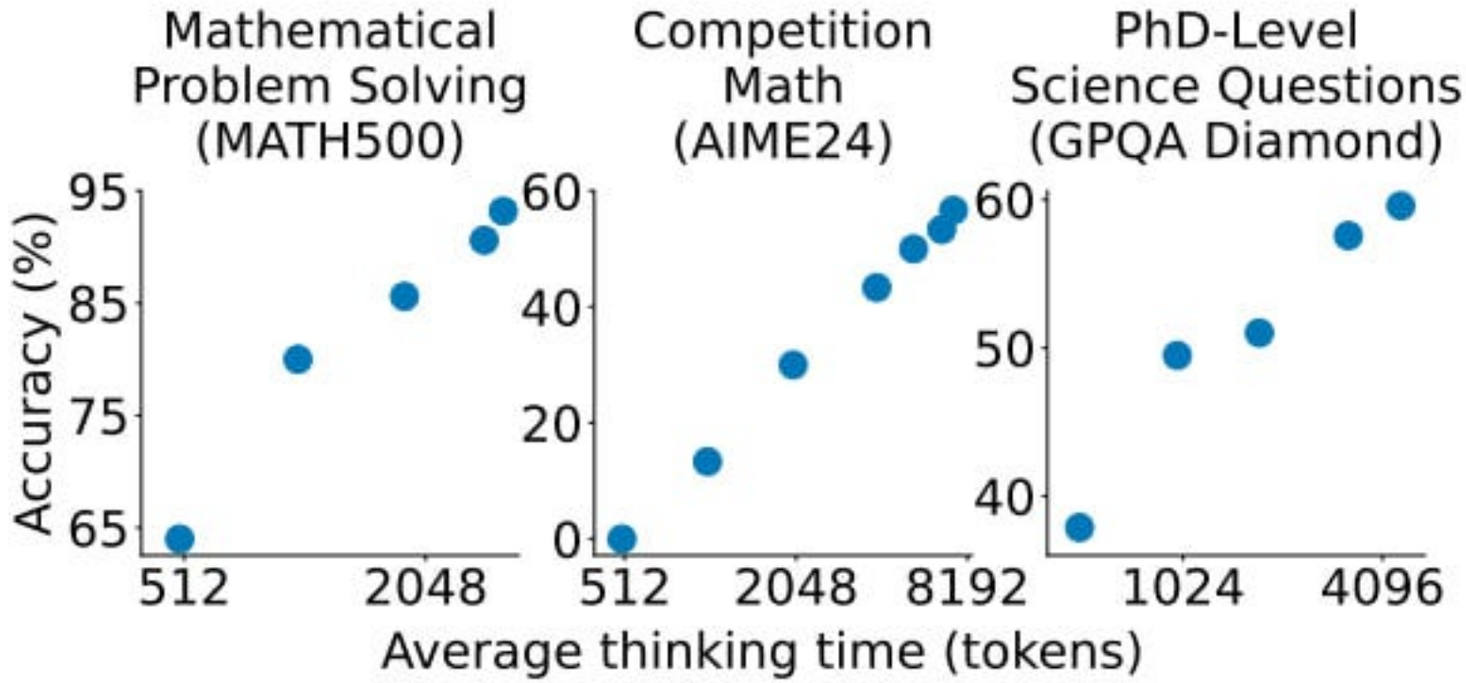
```
Show your reasoning explicitly using: <think>...</think>
Then provide:
Final: ...
Problem: {YOUR_PROBLEM}
```

- **Hidden vs shown CoT:** systems may use internal reasoning but only display a concise final explanation to users. Indeed, showing full reasoning can leak sensitive info, internal policies, or prompt details.

- **CoT vs test-time search:** CoT is one reasoning trace; search runs multiple traces/ candidates and selects/aggregates (*next slide*).

- **Evaluation:** CoT text can be convincing but wrong; you often need **independent checks** (tools, unit tests, constraints) rather than trusting the rationale.

The "Wait" Token: Simple Test-Time Scaling



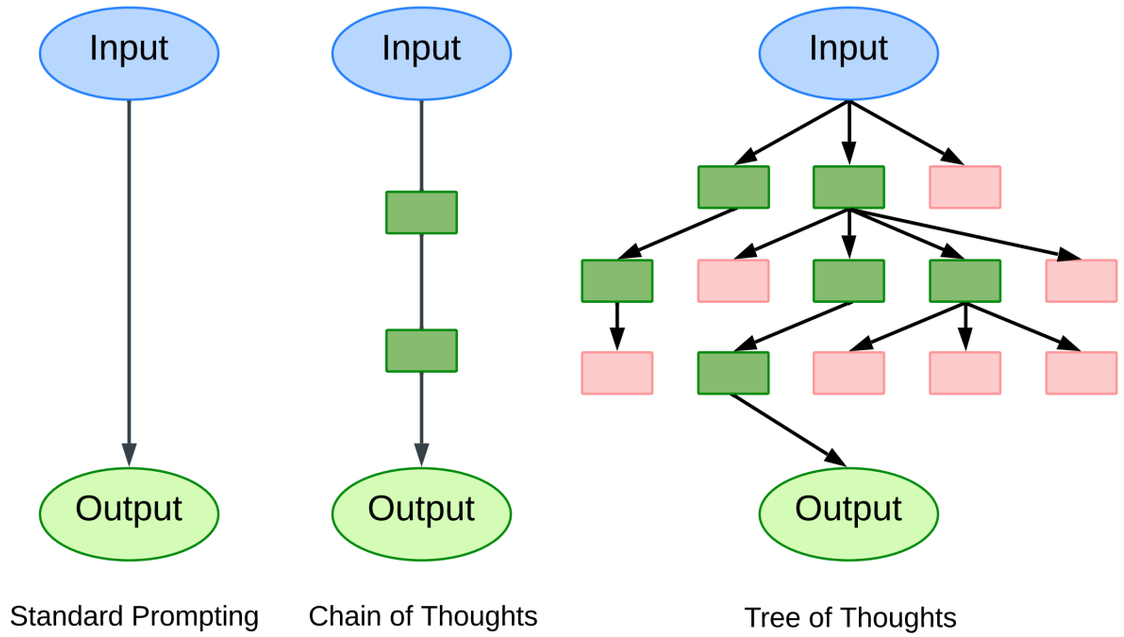
- **Idea:** improve reasoning at inference by **forcing extra "thinking" tokens** when the model tries to stop early ("test-time scaling")

- **Mechanism ("budget forcing"): suppress the end-of-thinking delimiter**; if the model attempts to end, append **"Wait"** to the ongoing reasoning trace to trigger revision

- "Wait" acts like injected doubt / hesitation, prompting the model to re-check and sometimes self-correct

- **Control knob:** you can enforce **minimum/maximum thinking length** (extend by repeated "Wait", or terminate when over budget).

Tree of Thoughts: Search Over Reasoning Paths



- **Idea:** instead of one chain-of-thought, generate **multiple intermediate "thoughts" (chunk of the think reasoning)** and explore them like a **search tree**.

- **Process: expand → evaluate → select/prune → repeat** until a solution is found (or budget runs out).

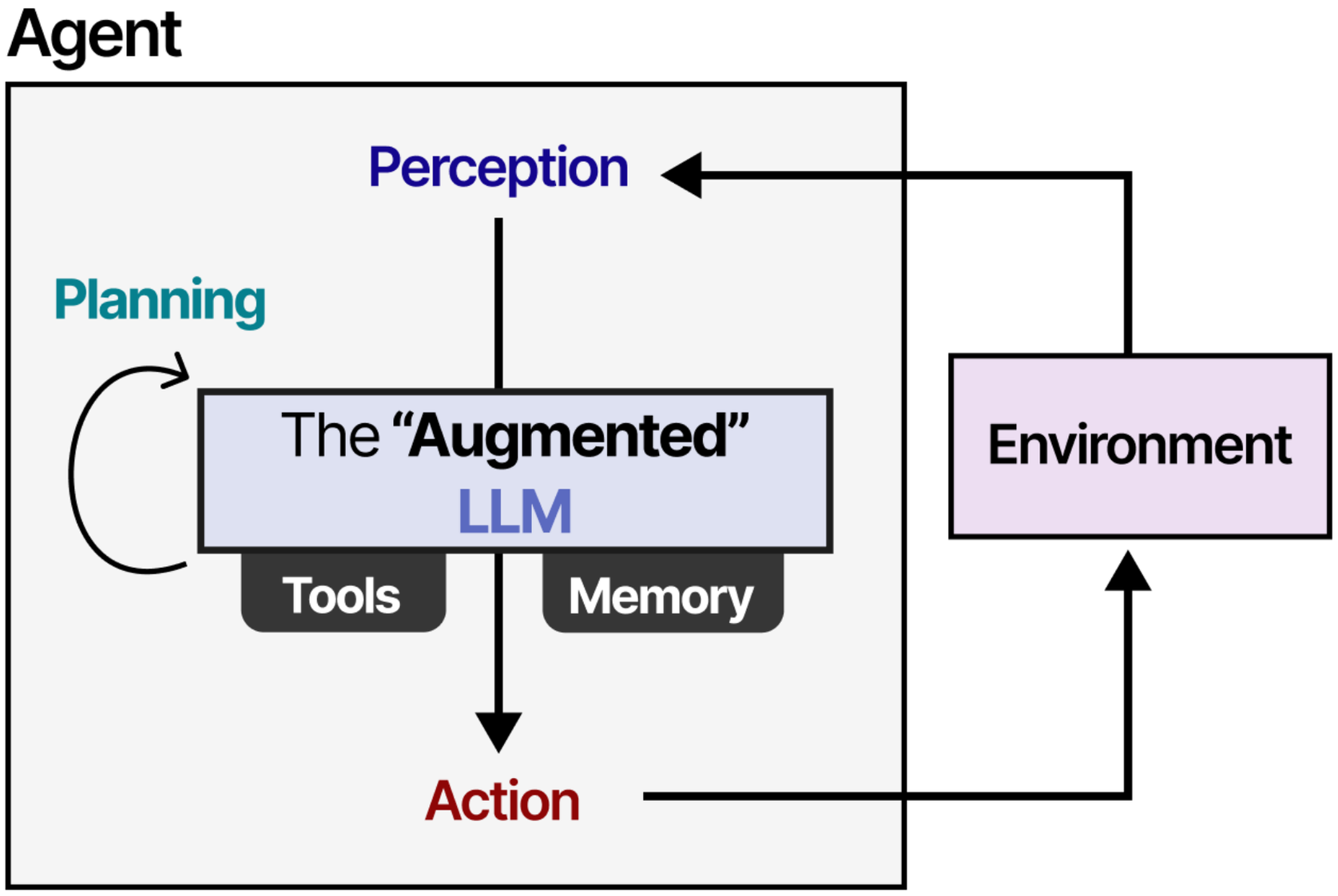
- **Search methods:** use **any tree search (e.g., beam search)** over candidate thoughts; keep the best branches.

- **Evaluation of thoughts:** score partial solutions via a **value function** (LLM-as-judge), heuristics, or external checks/tools.

- **Tradeoffs:** higher **latency/cost**, needs good scoring to avoid "wandering"; mitigate with tight budgets, pruning, and verification. *For now, this promising strategy does not deliver improvements.*

Smart Agents

Agents


Agent
Perception
Planning
The "Augmented"
LLM
Tools
Memory
Action
Environment

LLMs + tools (agents)

- **What "tools" are:** external functions/APIs an LLM can call to extend capabilities beyond its weights (act, look up, compute).

- **Common tool types: Search/Retrieval** (web, vector DB), **Datastores** (SQL, docs), **Compute** (Python, calculators), **Apps** (email/calendar/CRM), **Code** (repo access, tests), **Multimodal** (vision/OCR).

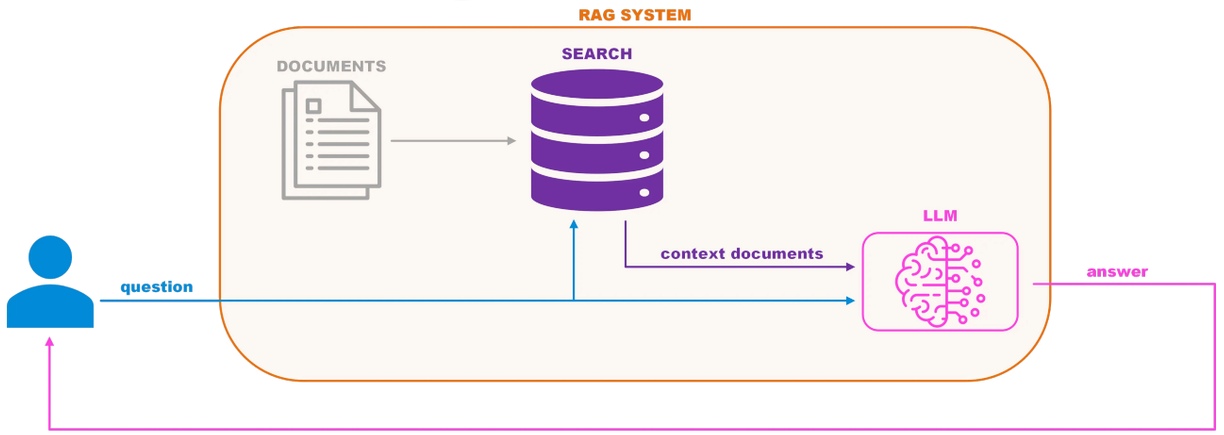
- **Loop of tools use:** Plan → Call tool → Observe result → Update → Answer, sometimes with multiple iterations.

- **Tool selection & routing:** choose the right tool via prompts/policies; use schemas (function signatures) and constrained outputs (JSON) to reduce errors.

- **Reliability guardrails:** input validation, permissions, rate limits, sandboxing, citation/trace logs, and fallback to "ask user / I don't know".

- **Tradeoffs:** better accuracy + actionability vs higher latency/cost and new failure modes (wrong tool, bad queries, tool errors).

Retrieval Augmented Generation



- **RAG?** ground answers in curated sources (less LLM hallucinations) + enable citations without retraining.

- **RAG = Retrieval-Augmented Generation:** retrieve relevant documents/chunks and feed them to an LLM to ground the answer.

- **Indexing (offline):** split docs into overlapping chunks, create embeddings&metadata, stored in a database.

- **Retrieval (online):** embed the user query: fetch top-K candidates and re-rank candidates.

- **Merge into prompt:** select top N tokens within a token budget; order by relevance and format as clearly labeled sources (S1, S2. . .) for citations. Generation can **marginalize over retrieved docs** using retrieved weights yet most production RAG simply concatenates selected chunks into the context.

- **Generation + guardrails:** LLM answers using provided sources, cites evidence; refuse/flag if evidence is weak; optionally verify claims against retrieved text.

Inference Stack

vLLM: High-Throughput LLM Serving

- **What it is:** open-source inference/serving engine focused on **high throughput** and **efficient GPU memory** usage.

- **Core idea: PagedAttention** manages the KV cache in **blocks** (paging-style) to reduce fragmentation and pack more concurrent requests

- **System design: preemptive scheduling**, co-designed with PagedAttention for better utilization under load

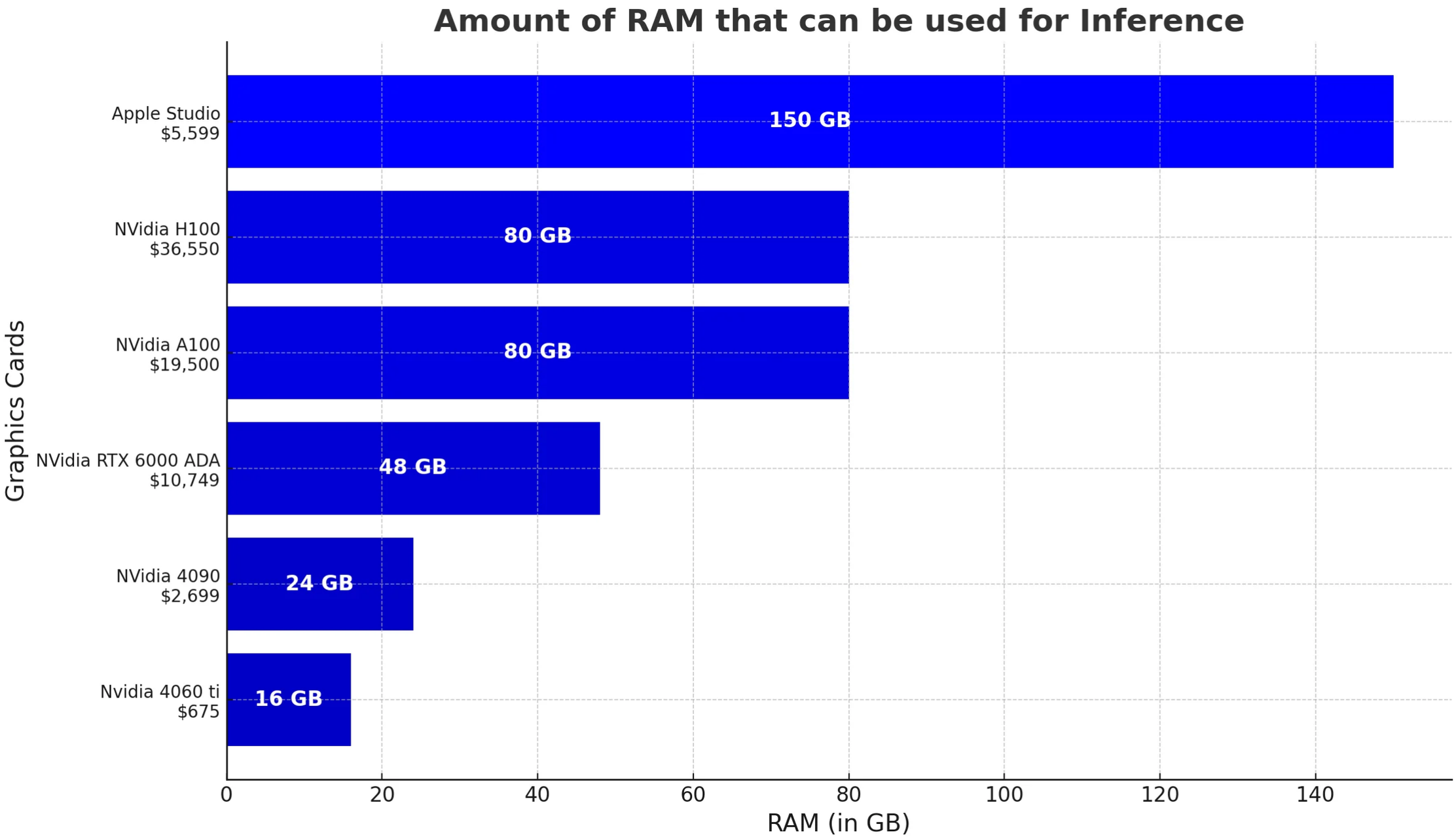
- **Why it's used in practice?** Strong production fit for **streaming generation** and continuous batching; widely adopted and actively developed

- **Serving interface:** ships an **OpenAI-compatible HTTP server** (Chat/Completions, etc.), so many clients can talk to it with minimal change

- **State model:** application-level stateless (you pass the context each call), but runtime state exists per request via the KV cache during generation

- **Key benefit:** more concurrency for a given VRAM budget $\rightarrow$ better **tokens/sec** and often better tail latency at high load

Hardware



The race is cruel... and surprising.

Conclusion

- Let's try out vLLM!
- The next lecture will be hosted by Dr. Michael Eickenberg, working right now at *H Company*.