The Lifecycle of a Language Model

Lecture 2 Notes

CDSS 94: Building Thoughtful AI Systems (Spring 2026)

Contents

1 Overview: Two Lifecycles

An LLM operates at two timescales that differ by roughly eight orders of magnitude.

Definition 1.1: Model Lifecycle vs. Request Lifecycle

- **Model Lifecycle:** Pretraining, post-training, deployment. Happens *once* per model. Costs \$10M–\$100M+. Takes months. *Creates* capabilities.
- **Request Lifecycle:** Tokenize, prefill, decode, stream. Happens *millions of times per day*. Costs fractions of a cent. Takes seconds. *Uses* capabilities.

Remark 1.2

If you want to build AI products, optimize AI systems, or think clearly about what AI can and can't do, you need to understand both lifecycles. Decisions made during the model lifecycle *constrain everything possible* in the request lifecycle.

1.1 The Model Lifecycle

The model lifecycle consists of three phases:

	Pretraining	**Post-Training**	**Deployment**
Purpose	Creating capabilities	Aligning & specializing	Serving at scale
Method	Next-token prediction on trillions of tokens	SFT, RLHF on curated datasets	Quantization, parallelism, API integration
Cost	\$10M – \$100M+	\$10K – \$1M	Fractions of \$/query
Duration	Months of compute	Days to weeks	Ongoing

Remark 1.3

Each model version undergoes pretraining *once*, baking in knowledge and capabilities that are reused during inference. Processing an LLM query is estimated to be about 10× more expensive than a keyword search query, making efficient serving crucial at scale.

2 The Hardware Lottery

2.1 The Core Argument

Definition 2.1: The Hardware Lottery (Hooker, 2020)

Research ideas succeed or fail not because they are inherently better or worse, but because of whether they happen to **fit the available hardware**.

Neural networks were proposed in the 1950s and languished for decades. Then GPUs became good enough at matrix multiplication, and suddenly deep learning worked. Every architectural decision in modern LLMs—attention mechanisms, feedforward networks, specific dimensions—is optimized for NVIDIA GPUs. Not because that is mathematically optimal, but because that is what we have.

Remark 2.2: Uncomfortable Implication

There may be better architectures that we will never discover because they don't fit our hardware. The hardware lottery is still running. Every LLM you use today is shaped by this lottery.

2.2 The Memory Wall

Theorem 2.3: The Memory Wall — informal

Between 2016 and 2024, NVIDIA scaled compute (raw FLOPS) by $\sim 100\times$, but memory bandwidth only scaled $\sim 11\times$ and memory capacity only scaled $\sim 12\times$.

The consequence: GPUs can compute $100\times$ faster, but can only be fed data $\sim 10\times$ faster. The GPUs are *starving*.

> *When people say LLM inference is "memory-bound," this is what they mean. The compute units are sitting idle, waiting for data to arrive from memory. It's not that the math is slow—the math is basically instant. It's that we can't get the data to the math fast enough.*

This single fact explains almost every optimization technique in LLM inference: batching (do more useful math while data is already loaded), FlashAttention (avoid unnecessary memory trips), KV caching (skip re-loading data already seen), and streaming (each token requires loading billions of parameters from memory).

3 Arithmetic Intensity

3.1 Definition

Definition 3.1: Arithmetic Intensity

Arithmetic intensity is the ratio of compute operations to memory accesses:

$$\text{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Bytes loaded from memory}}$$

A workload is **compute-bound** if its arithmetic intensity exceeds the hardware's crossover point, and **memory-bound** if it falls below.

Example 3.2: Kitchen Analogy

Imagine you are a chef with a cutting board (compute/tensor cores) and a walk-in fridge down the hall (HBM/main memory).
Arithmetic intensity asks: *for each trip to the fridge, how many cuts do you make?*
Low arithmetic intensity: grab one onion, make one cut, walk back to the fridge. You spend all your time walking, not cooking. The fridge trips are the bottleneck.
High arithmetic intensity: grab twenty onions, make a hundred cuts before your next trip. You spend most of your time actually cooking. The cutting speed is the bottleneck.

3.2 The GPU Reality

For an NVIDIA H100 GPU, the crossover point is approximately **200** ops/byte.

Regime	**Arithmetic Intensity**	**Bottleneck**
H100 crossover point	~ 200 ops/byte	—
LLM decode	~ 1–10 ops/byte	Memory-bound

LLM decode sits 20–200× below the crossover point. We are spending $\sim 95\%$ of the time waiting for data, not computing.

3.3 The Core Asymmetry: Prefill vs. Decode

Definition 3.3: Prefill vs. Decode

- **Prefill:** Process the full prompt at once. Many tokens per weight load. High arithmetic intensity. **Compute-bound.** High GPU utilization.
- **Decode:** Generate one token at a time. ~ 1–10 ops per byte loaded. Low arithmetic intensity. **Memory-bound.** $\sim 95\%$ of time spent loading weights.

Remark 3.4

The game of efficient inference is: *keep data in the fastest memory possible.* Every clever optimization technique is a different answer to this same question.

4 Pretraining Decisions That Shape Inference

Decisions made during pretraining are *baked into the weights*. You cannot change them later without retraining from scratch. Three key decisions constrain all downstream inference.

4.1 Attention Variants and the KV Cache

During generation, the model looks back at all previous tokens using Queries (Q), Keys (K), and Values (V):

Definition 4.1: Attention Mechanism (Q, K, V)

- **Query (Q):** "What am I looking for right now?" — comes from the current token. *Changes every token.*
- **Key (K):** "What does each previous token have to offer?" — like labels or tags. *Remains the same* once computed.
- **Value (V):** "What information does each previous token actually contain?" — the actual content. *Remains the same* once computed.

Since K and V for previous tokens don't change, we **cache** them. This is the **KV cache**.

Proposition 4.2: KV Cache Size

$$\text{KV cache} = 2 \times n_{\text{heads}} \times L_{\text{seq}} \times d_{\text{head}} \times b_{\text{bytes}}$$

where n_{heads} is the number of KV heads, L_{seq} is the sequence length, d_{head} is the head dimension, and b_{bytes} is the bytes per parameter.

Example 4.3: KV Cache Scaling

For a 70B model with 64 KV heads, at 8K context in FP16: $\sim$5 GB per request. At 128K context: $\sim$80 GB per request—an entire H100's memory for *one* user's conversation.

4.2 Attention Variants

The number of KV heads is a pretraining architectural choice:

Variant	**KV Heads**	**Tradeoff**
Multi-Head Attention (MHA)	64 Q $\rightarrow$ 64 KV pairs	Maximum quality, maximum memory
Grouped-Query Attention (GQA)	64 Q $\rightarrow$ 8 KV pairs	Near-same quality, 8$\times$ less memory
Multi-Query Attention (MQA)	64 Q $\rightarrow$ 1 KV pair	Quality loss, minimum memory

Remark 4.4: Baked-In Choice

You cannot switch from MHA to GQA after training. "Uptraining" (the GQA paper) requires $\sim$5% of pretraining compute—still millions of dollars for frontier models. GQA vs. MHA can mean the difference between fitting 10 concurrent users in memory versus 80.

Remark 4.5: Multi-head Latent Attention (MLA)

DeepSeek-V2 introduced MLA, which compresses Keys and Values into a smaller latent representation before caching. Like storing ZIP files instead of raw files. Smaller cache, cheap decompression, near-zero quality loss. Even more deeply baked into the architecture—cannot be retrofitted.

4.3 Vocabulary Size

Definition 4.6: Vocabulary Size Tradeoffs

Larger vocabulary = fewer tokens per sequence = faster, cheaper inference. But also: larger embedding table (∼2 GB at 128K vocab × 8192-dim × FP16) and a bigger softmax computation on *every generated token*.

Example 4.7: Tokenization Across Languages

Language	**Tokens per Word**
English	~ 1.3
Chinese	~ 2–3
Korean	~ 3–4

Non-English users pay more for the same semantic content. This is not just an efficiency issue—it is an equity issue.

4.4 Context Length

Models are often trained on much shorter contexts (4K–8K tokens) than they are served with (128K+).

Definition 4.8: RoPE — Rotary Position Embeddings

RoPE encodes relative positions through rotation matrices, enabling generalization beyond training context length. Extensions include RoPE scaling, NTK-aware interpolation, and YaRN.

Remark 4.9

Limits persist: the KV cache still grows linearly with context, attention computation still grows, and quality degrades at lengths far beyond the training distribution.

5 How Post-Training Affects Inference

5.1 The Post-Training Stack

Pretraining gives you next-token prediction (expensive autocomplete). Post-training turns it into an assistant. The standard stack is:

$$\text{Pretrained Model} \longrightarrow \text{SFT} \longrightarrow \text{Preference Optimization} \longrightarrow \text{Safety Layers}$$

Each stage changes the weights in ways that affect inference cost and behavior.

5.2 Supervised Fine-Tuning (SFT)

SFT fine-tunes the pretrained model on examples of desired behavior: the conversational format, when to stop generating, the pattern of question $\rightarrow$ answer $\rightarrow$ end-of-sequence.

Remark 5.1: Inference Implication

If you don't format your prompt correctly, you lose most of the benefit of SFT. Every model has its expected format (`[INST]`, `<|im_start|>`, etc.). A single changed space or newline can break production systems. The model is pattern matching—you must give it the patterns it learned.

5.3 RLHF vs. DPO

SFT teaches format. Preference tuning teaches *quality*.

Definition 5.2: RLHF

Train a separate reward model to predict human preferences, then use PPO to optimize the language model against it. Requires **4 models in memory**: policy, reference, reward, and value model. Powerful but unstable—prone to reward hacking.

Definition 5.3: DPO (Rafailov et al., 2023)

The language model *is* the reward model. Derives a loss function that directly optimizes the policy on preference pairs, no RL required. The optimal policy under RLHF can be written as a function of the policy itself and a reference policy:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}\left[\log \sigma\left(\beta \cdot \left(\log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\right)\right)\right]$$

Simpler, more stable, same objective.

5.4 GRPO: Group Relative Policy Optimization

Both RLHF and DPO need human-labeled preference pairs. For *verifiable* domains (math, code), we can skip human labeling entirely.

Definition 5.4: GRPO (DeepSeek)

Given a prompt (e.g., a math problem):

1. Generate a **group** of N responses (e.g., 8 attempts).
2. **Verify** which are correct (check the answer).
3. Correct answers become "preferred"; incorrect become "not preferred."
4. Train on these **outcome-based pairs** using relative comparison within the group.

No human labeling needed. Scales to millions of training pairs.

Remark 5.5: Why GRPO Matters

- **Scale:** Millions of pairs vs. hundreds of thousands from human labeling.
- **Cost:** DeepSeek spent a fraction of what OpenAI spent on RLHF.
- **Quality for reasoning:** Outcome-based signal avoids the bias where humans prefer confident-sounding wrong answers over hesitant correct ones.
- **Limitation:** Only works where you can verify the answer. For open-ended tasks ("write a poem"), you still need preference signal.

Remark 5.6: The Post-Training Portfolio

In practice, frontier labs likely use all three:

- GRPO for math, code, and reasoning tasks
- DPO for general helpfulness and style
- RLHF (possibly) for tricky edge cases

The recipe is a portfolio matched to different aspects of behavior.

5.5 System Prompts: The Hidden Cost

Every request includes a system prompt: personality, guidelines, safety instructions—often 2000+ tokens.

Example 5.7: System Prompt Cost

If the system prompt is 2000 tokens and the user sends 10 tokens, then 99.5% of prefill compute is spent on the system prompt. This happens on *every single request.*

Definition 5.8: Prompt Caching

Cache the KV values from the system prompt across requests. Subsequent requests reuse the cache, skipping redundant prefill. Requires infrastructure support and adds complexity, but can dramatically reduce per-request latency.

5.6 Safety at Inference Time

A significant portion of safety enforcement happens at inference time, not training time:

(i) **Input classifiers:** Block or flag harmful requests *before* the model runs.
(ii) **Output classifiers:** Filter personal information and harmful content from generated responses.
(iii) **Model-as-judge:** Generate a response, then self-check against guidelines.

(iv) **Constitutional AI:** Self-critique built into the generation process itself.

Remark 5.9

All of these add latency and cost. Safety is a computational tax on every request—but the alternative is worse.

6 The Request Lifecycle

6.1 Latency Metrics

Three metrics define the user experience of LLM inference:

Definition 6.1: TTFT, ITL, End-to-End Latency

- **TTFT (Time to First Token):** How long from hitting enter until the first character appears. Dominated by prefill time + queueing delay. Target: < 1–3 seconds.
- **ITL (Inter-Token Latency):** Time between each subsequent token. How fast the response "types." Target: 30–100 ms.
- **End-to-end latency:**

$$\text{Latency} = \text{TTFT} + (\text{output\_length} \times \text{ITL})$$

Example 6.2: Latency Calculation

A 500-token response with TTFT $= 1$ s and ITL $= 50$ ms:

$$1 + 500 \times 0.05 = 26 \text{ seconds}$$

Remark 6.3

There is a real tradeoff between TTFT and ITL. Optimizing purely for TTFT (process requests immediately) yields small decode batches and high ITL. Optimizing for ITL (maintain full batches) makes requests wait in queue, increasing TTFT. Good serving systems balance these via techniques like continuous batching.

7 Summary: Unifying Principles

Memory, not compute, is the bottleneck. The whole field of LLM optimization is: how do we work around the fact that memory can't keep up with compute?

The complete token lifecycle reveals several unifying principles:

(i) **Memory, not compute, dominates autoregressive decoding.** Optimizations must reduce HBM traffic or amortize weight loading across larger batches.

(ii) **The memory hierarchy is exploitable.** FlashAttention's 16× HBM reduction comes from keeping computation in SRAM.

(iii) **The systems stack is compositional.** FlashAttention handles attention compute, PagedAttention manages memory allocation, continuous batching schedules requests, speculative decoding parallelizes generation. Production systems combine all of these.

(iv) **Hardware constrains algorithmic success.** The Hardware Lottery reminds us that transformers dominate partly because they match GPU matrix-multiply primitives. Future architectures—linear attention, state-space models, or entirely new approaches—will succeed or fail based on hardware fit, not just theoretical properties.

(v) **Pretraining decisions echo into every inference call.** Attention variant, vocabulary size, context length—these choices are baked into the weights and constrain all downstream serving.

Optimization	**Core Idea**
Batching	Amortize weight loading across multiple requests
FlashAttention	Keep computation in SRAM, avoid HBM round-trips
KV Cache	Don't re-compute Keys/Values for previous tokens
PagedAttention	Virtual memory paging for KV cache management
Continuous Batching	Insert new requests without waiting for a full batch to finish
Speculative Decoding	Guess multiple tokens with a small model, verify in parallel
Prompt Caching	Reuse KV values from shared system prompts
Quantization	Reduce precision to lower memory and bandwidth requirements

Understanding the full stack—from GPU architecture to production API—is essential for building the next generation of efficient AI systems.