

CMSC828G: Special Topics in Systems for Machine Learning

LLM Inference Systems

Instructor: Yafan Huang



DEPARTMENT OF
COMPUTER SCIENCE

Logistics: Updated Schedule and AI Usage

- Final Project Pitch Presentations: Oct. 20 (Tue) -> Oct. 27 (Tue)
- Attention Approximation: Oct. 27 (Tue) -> Oct. 20 (Tue)
 - Reading assignments in ELMS will be updated accordingly.
- Questions about final projects?
 - AI Usage

Content

- Why inference serving is a systems problem?
- Fundamentals: prefill, decode, KV cache, scheduling
- Core optimizations: chunked prefill, prefix caching, speculation
- Benchmarking and metrics
- (optional) Framework landscape: vLLM, SGLang, TensorRT-LLM, Dynamo, etc.

Terminology: In this lecture, *inference systems* and *LLM serving systems* are used interchangeably unless otherwise specified.

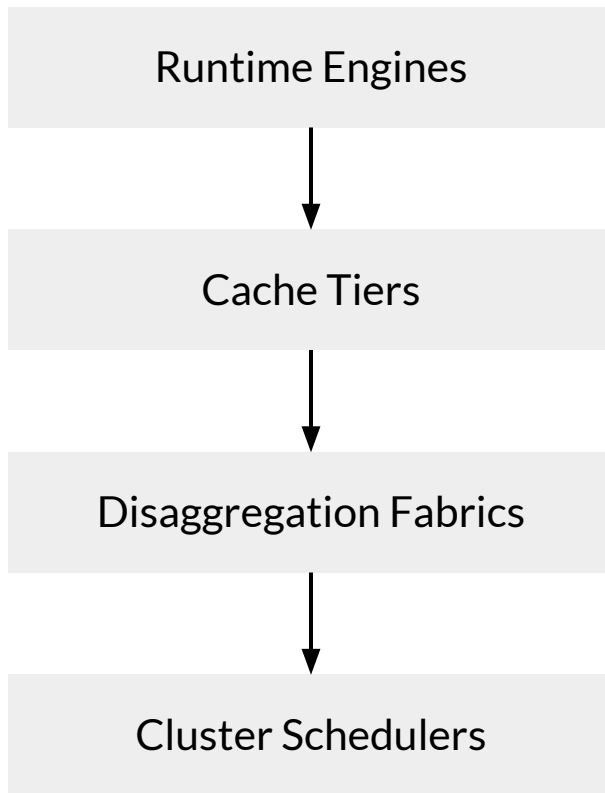
Why inference serving is a systems problem?

The serving problem has changed

- LLM inference system: from “single-turn chat on a dense model” serving to reasoning models, agentic workflows, and multimodal models serving.
- These workloads create new asymmetries: prompts are long and requests frequently revisit old context or tool traces.
- Memory movement, KV reuse, and routing matter as much as raw kernel speed.
- The serving plane has split into layers: runtime engines, cache tiers, disaggregation fabrics, cluster schedulers.

“The model” is only one piece of the product.
The real system is an **inference factory**.

Modern Inference System - Layers



Attention kernels • Continuous batching • KV management
Examples: vLLM, SGLang, TensorRT-LLM

GPU HBM • CPU DRAM • Remote memory • SSD
Examples: LMCache, HiCache

Prefill/decode separation • KV transfer • Network transport
Examples: NIXL, RDMA-based transport

Request routing • GPU placement • Load balancing
Example: Dynamo router

Modern Inference System - Features

Traffic Characteristics

- Long system prompts and retrieved context
- Multi-turn chat histories
- Bursty enterprise traffic with diurnal patterns
- Large MoE and reasoning models (CoT)

What This Stresses

- Time to first token under long prompts
- Decode latency under high concurrency
- GPU memory for KV state
- Load balancers that ignore cache locality
- Tail latency, not just median

Engineering Consequences

- Continuous batching is table stakes
- Prefix caching / prompt reuse are economically important
- Prefill and decode are separated or scheduled differently
- MoE models require expert parallelism + DP-attention + fast all-to-all(v)
- Framework choice depends on flexibility vs. portability vs. peak hardware performance

Fundamentals

The two phases of autoregressive inference

- Every request goes through two distinct phases:

Prefill (prompt processing)

- Process all input tokens in parallel
- Compute and store key / value (KV) states for each layer and each attention head
- Typically **compute-bound**
- Dominates **time to first token (TTFT)**

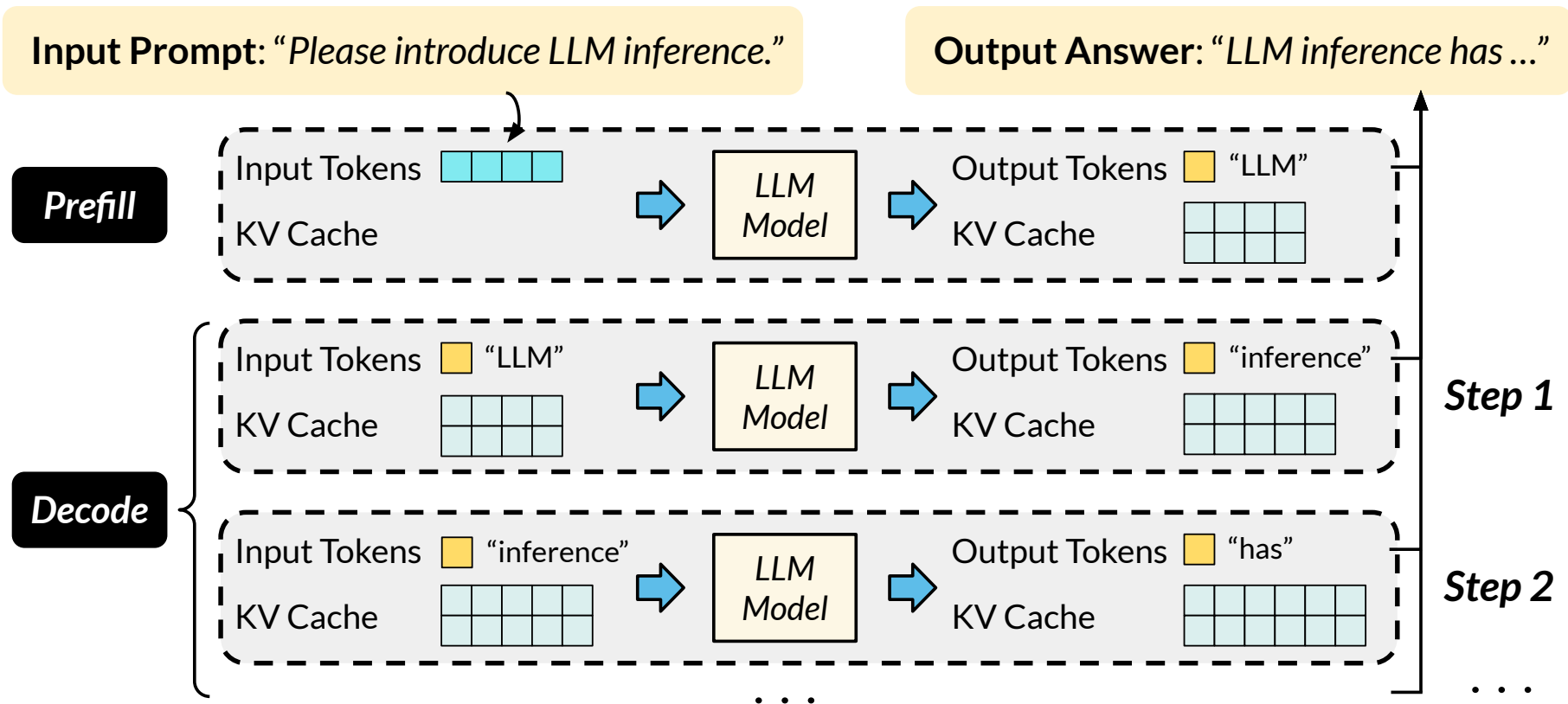
Decode (token generation)

- Generate one new token per step
- Reuse stored KV instead of recomputing
- Typically **memory-bandwidth-bound**
- Dominates **inter-token latency (ITL)** and streaming smoothness
- The bigger pain point at high concurrency

Different compute profiles -> different batching policies, parallelism choices, and sometimes different hardware.

This asymmetry drives most of the design decisions we will see today (e.g., AWS Trainium and Cerebras WSE).

The two phases of autoregressive inference



$$\text{ITL} = (\text{E2E Latency} - \text{TTFT}) / (\text{Total Output Tokens} - 1)$$

Inference latency: where does the time go?

- $TTFT = t_{\text{queue}} + t_{\text{tokenize}} + t_{\text{prefill}} + t_{\text{sample}} + t_{\text{first-byte}}$
- $ITL \approx t_{\text{decode step}} + t_{\text{sample}} + t_{\text{stream flush}}$
- System-level Metrics:
 - Output throughput: output tokens/s across the fleet
 - Useful work / GPU: tokens/s/GPU, or revenue-generating tokens per GPU-hour
 - SLO compliance: p50 / p95 / p99 targets by traffic class
 - Cache efficiency: prefix-hit rate, reusable-KV hit rate
 - Admission stability: queue depth, rejection rate, timeout rate

p99 (the 99th percentile) means that 99% of requests or events finish faster than this value, while the slowest 1% take longer.

Why decode is the harder systems problem

- Each decode step does little arithmetic relative to the model and KV state it must touch.
- The per-token path is therefore sensitive to:
 - HBM bandwidth
 - inter-GPU synchronization (TP/EP)
 - host overheads (scheduler, grammar checks, stream flush)
 - queueing interactions with prefills
- A system with excellent kernel TFLOP/s can still feel slow if decode p99 is unstable.

In practice, users rarely complain “*my decode throughput is low.*” They complain “*my stream stalled,*” or “*some requests are 10x slower than others.*”

KV Cache

- For one sequence context length T : $KV_{\text{bytes}} \approx T \cdot 2 \cdot L \cdot H_{\text{kv}} \cdot d_h \cdot b$
 - L = number of transformer layers
 - H_{kv} = number of key/value heads
 - d_h = head dimension
 - b = bytes per element (KV datatype)
 - factor of 2 = keys + values
- For a batch: replace T with the total number of cached tokens across active requests.

KV Cache: workload example

- Suppose a model has: $L = 80$, $H_{kv} = 8$, $dh = 256$, $b = 2$ bytes
- Then KV bytes per token: $2 \cdot 80 \cdot 8 \cdot 256 \cdot 2 = 655,360$ bytes ≈ 640 KiB/token
- So
 - 8K tokens for one request ≈ 5.0 GiB of KV
 - 32 concurrent requests at 8K context ≈ 160 GiB of KV state
 - This is why engines invest in paged allocation, reuse, eviction, and offload
- If your workload is long multi-turn sessions, the memory system is your real product surface.

Continuous Batching vs. Static Batching

Static batching

- Form a batch, wait for all to finish together
- Suffers when requests have different lengths
- Wastes GPU time on padding

	T1	T2	T3	T4
Slot 0	A	A	A	A
Slot 1	B	B	B	
Slot 2	C	C		

Continuous batching

- Admit new work whenever token budget and KV space allow
- Requests can be at different generation steps
- Keeps hardware busier, reduces idle slots
- Needs tracking for sequence lengths, pages, masks

	T1	T2	T3	T4
Slot 0	A	A	A	A
Slot 1	B	B	B	C
Slot 2	D	D	E	E

Scheduler Responsibilities

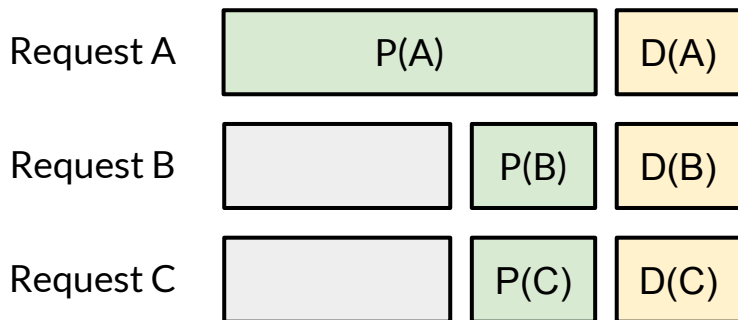
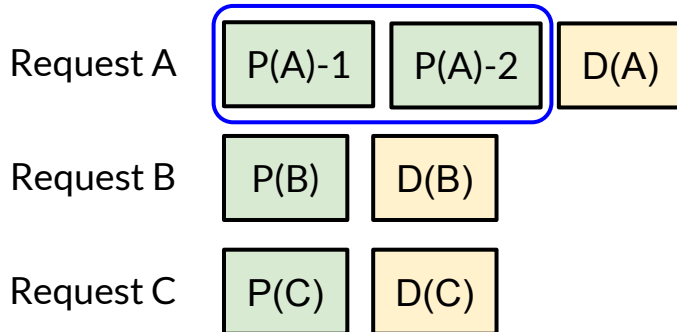
- Scheduler: A runtime component that decides which requests to execute at each iteration, subject to compute and memory constraints.
- A modern scheduler decides, per engine step:
 - Which waiting requests may be admitted
 - How much token budget goes to decode vs. prefill
 - Whether long prefills should be chunked
 - Whether speculative draft tokens fit in the token / page budget
 - Whether reusable or offloaded KV should be reloaded
 - Which requests to preempt or deprioritize under pressure

Core Optimizations

Chunked Prefill

- Motivation: Long prompts can delay the decoding of other requests.
- **Chunked prefill** splits the prompt into smaller pieces so the scheduler can interleave prefills with decode steps.
- In vLLM V1, chunked prefill is enabled by default. Decode requests have higher priority, and prefill uses the remaining token budget.

Chunked and interleaved with other works.



Chunked Prefill

- Motivation: Long prompts can delay the decoding of other requests.
- **Chunked prefill** splits the prompt into smaller pieces so the scheduler can interleave prefills with decode steps.
- In vLLM V1, chunked prefill is enabled by default. Decode requests have higher priority, and prefill uses the remaining token budget.
- Tradeoff
 - Smaller chunks: More scheduling overhead, but lower ITL.
 - Larger chunks: Less scheduling overhead, but higher ITL.

Decode step interleaved with prefill chunks!

Prefix Caching

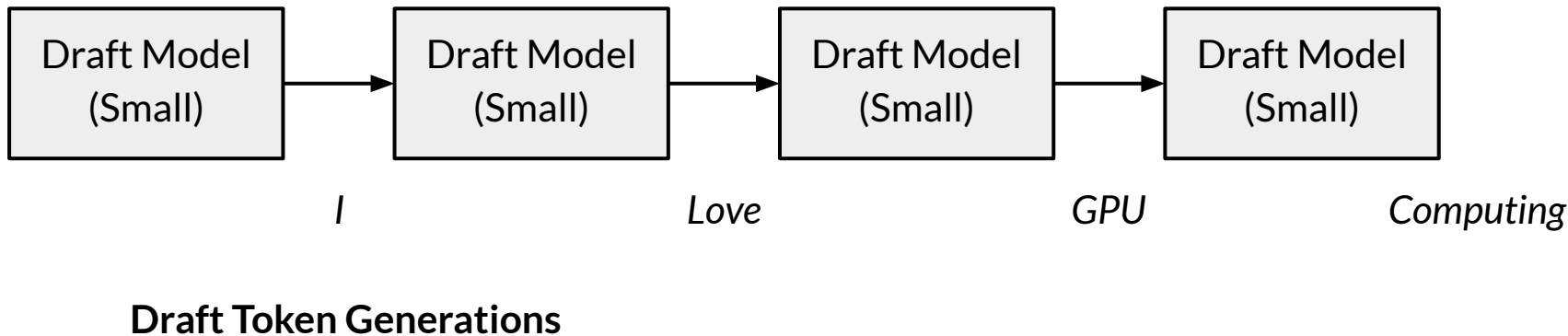
Token IDs → Prefix Hash Matching → KV Cache Reuse

- If two requests share a prompt prefix, the engine can reuse previously computed KV for the shared part. How it works: Match token prefixes using **block hashes** and reuse the corresponding KV cache.
- This reduces prefill cost and therefore TTFT; it does not speed up decode.
- Works well for:
 - shared system prompts,
 - repeated queries over the same document,
 - multi-round conversations,
 - agent loops (especially coding agents) that re-emit large structured context.

Speculative Decoding

- A **smaller draft model** proposes multiple tokens; the **larger target model** verifies them in fewer expensive steps.
- Helps most when decode is the bottleneck and draft predictions are accurate.

Draft Inference (multiple pass in a decoding loop)

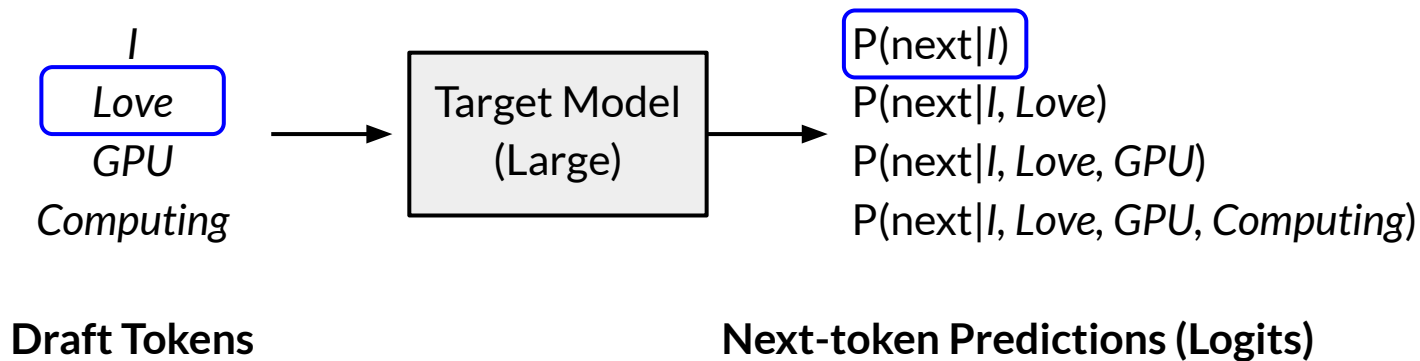


Speculative Decoding

- A **smaller draft model** proposes multiple tokens; the **larger target model** verifies them in fewer expensive steps.
- Helps most when decode is the bottleneck and draft predictions are accurate.

Target Inference (one pass with multiple tokens)

Target Model Verification



Quantization in Inference

- Quantization can target **weights, activations, and KV cache**.
- The right question is not “what is the lowest precision?” but:
 - what precision does my hardware support,
 - what kernels are mature,
 - what accuracy loss is acceptable,
 - whether memory savings actually convert into higher concurrency or cheaper hardware.
- TensorRT-LLM supports FP8 on Hopper and FP4 on Blackwell; vLLM and SGLang offer broader quantization menus across more hardware.

Benchmarking and Metrics

How to benchmark inference frameworks

- Use a realistic request mix: prompt lengths, output lengths, concurrency...
- Distinguish **throughput mode** from **latency-constrained mode**. (how?)
- Measure both **per-request metrics** (p50/p95/p99 TTFT, ITL) and **system-level metrics** (tokens/s/GPU, achieved concurrency, rejection rate).
- Warm caches and graphs, but also benchmark cold-cache and cache-miss cases.
- Test the feature combination you will actually ship: quantization + guidance + speculation + PD, not each in isolation.

Throughput mode: How many tokens can the system generate per second?

Latency mode: How much throughput can the system achieve while meeting latency requirements?

Some Metrics to Track

- Admission / queueing: waiting requests, rejected requests...
- Latency: TTFT, prefill time, decode time, ITL, p95/p99 histograms
- KV memory: cache usage %, reusable blocks, offload / reload rates
- Workload shape: prompt length, output length, hit rate by prompt family
- Distributed health: all-to-all latency, transport errors, cross-engine KV transfer rate (identifies TP/EP/PD bottlenecks)
- Business efficiency: tokens/s/GPU, cost/request, cache-adjusted utilization

p99 latency: usually a scheduler or architecture problem

- Median performance can look great while p99 is terrible.
- Typical p99 killers:
 - long prompt admitted at the wrong time,
 - cache miss on a large prefix,
 - MoE expert imbalance,
 - speculative settings that overconsume token budget,
 - transport retries in PD mode,
- Most of these are addressed by chunked prefill, overlap, routing, and phase separation, not just by faster kernels.

Debugging: from symptom to likely subsystem

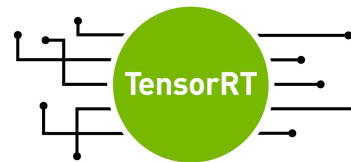
Symptom	Likely bottleneck	First questions
High TTFT, stable ITL	prefill, tokenization, or cache miss	Are prompts longer? Did hit rate fall? Prefill pools saturated?
Good medians, awful p99 ITL	scheduler interference or transport stalls	Long prefills interrupting decode? TP/EP collectives spiking?
GPU OOM / low concurrency	KV footprint / fragmentation / offload policy	Resident context exploding? Block size or reuse policy wrong?
Speculation gives no benefit	poor acceptance or scheduler pressure	What is acceptance rate? Draft tokens eating budget?
PD regresses performance	transfer / routing overhead	KV transfer fast enough? Requests placed with affinity?

(Optional) Framework Landscape

- Widely adopted open-source engine for high-throughput, memory-efficient LLM serving.
- Core ideas: **PagedAttention**, continuous batching, CUDA/HIP graph execution, chunked prefill, prefix caching, speculative decoding, multi-LoRA, broad hardware portability.
- Spans single-node to multi-GPU/multi-node with distributed scheduler support.
- Good default generally.
- Recent additions to the mainline: V1 engine with stronger scheduler, chunked prefill on by default, disaggregated prefill/decode guidance, EP + DP for MoE.

- High-performance serving framework for LLMs and multimodal models.
- Key features: **RadixAttention** for prefix caching, zero-overhead CPU scheduling, continuous batching, paged attention, TP/PP/EP/DP, structured outputs, quantization, multi-LoRA.
- Prominent for serving reasoning and MoE models at scale.
 - PD disaggregation as a first-class serving mode
 - Large-scale EP for MoE (challenge: communication and expert locality)
 - HiCache: hierarchical KV caching beyond GPU memory (Mooncake is its L3 storage backend)

TensorRT-LLM



- NVIDIA-optimized stack for LLM inference on NVIDIA GPUs.
- Features: **in-flight batching** (continuous batching), paged KV cache, overlap scheduling, disaggregated serving, speculative decoding, long-sequence support, LoRA, multimodal, quantization, AutoDeploy.
- Best fit when peak performance on Hopper / Blackwell is the priority. (not as portable as vLLM / SGLang)
- Moving toward a PyTorch backend + LLM API, reducing the gap between model definition and deployment.

NVIDIA Dynamo



- Distributed inference framework for **scaling** reasoning-model serving.
- Four design pillars:
 - GPU Resource Planner: moves capacity between prefill and decode pools
 - Smart Router: KV-aware request placement
 - Low-latency communication library for fast KV transfer
 - KV Cache Manager for multi-tier caching
- TensorRT-LLM optimizes inference execution **within GPU workers**, while Dynamo **coordinates multiple workers** for efficient distributed serving.

- Central idea: KV is a first-class systems object that can be stored, moved, reused, and shared across requests and engines.
- Two main use cases:
 - **Context caching:** cross-query reuse via offload / persistent storage
 - **PD disaggregation:** cross-engine KV transfer
- On long-context traffic, engineering KV economics often matters more than another kernel optimization.

Framework Comparison

Stack	Think of it as	Strengths	Typical fit	Watch-outs
vLLM	general open engine	portability, community, balanced features	research-to- production, many model families	may not be peak on one vendor path
SGLang	optimized serving stack	MoE / PD / gateway / structured output / HiCache	frontier deployments, agentic traffic	ecosystem evolving; large operational surface
TensorRT-LLM	NVIDIA-specialized runtime	peak NVIDIA perf, overlap scheduler, quantization	NVIDIA-only production	feature-combination caveats, less portable
Dynamo	distributed control plane	resource planning, smart routing, KV tiering	multi-node reasoning serving	needs a runtime engine underneath
LMCache	KV layer / extension	long-context reuse, offload, cross-engine KV	cache-heavy or PD systems	only helps if workload has reusable KV

Acknowledgement

- Parts of these slides were adapted from and inspired by materials from UC Berkeley's Scalable AI: Bridging Theory, Understanding, and Practice (Spring 2026), particularly [Lecture 18: Fundamentals and Overview of High-Performance Inference Frameworks](#). I gratefully acknowledge the course instructors and authors of these materials.
- Some figures were adapted from my previously published research papers.