Efficient LLM Inference: Autoregressive Decoding, Speculative Execution, Quantization, and Serving Stacks

October 25, 2025

1 Token-by-Token Generation

1.1 Autoregressive pipeline overview

Large language models follow an autoregressive decoding loop: given the prefix $x_{<t}$, the model predicts a distribution for the next token x_t, samples or chooses a candidate, and appends it to the context. The front-end embedding, stacked self-attention blocks, feed-forward networks, and output projection are executed at every step, while key/value (KV) cache stores historical states to avoid recomputing attention against the full sequence. Figure **??** highlights the canonical pipeline.

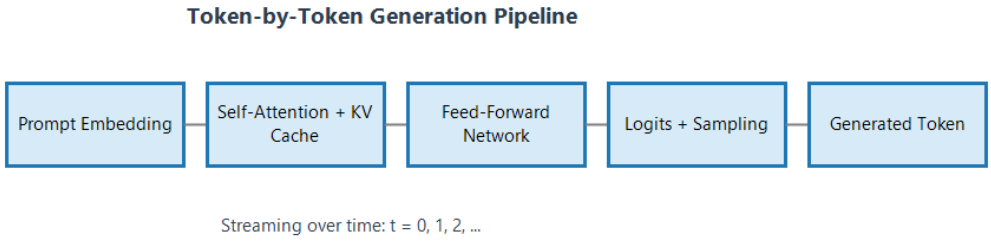

Figure 1: Token-by-token generation pipeline with KV cache reuse and per-step sampling.

Core properties:

- **Latency scales with length:** Each decoded token triggers a full forward pass, so response time grows linearly with the number of generated tokens.
- **State reuse is critical:** KV cache reduces attention complexity from $O(n^2)$ to $O(n)$ per step, trading memory for compute.
- **Sampling is decoupled:** Temperature, top-k, nucleus sampling, and repetition penalties are applied to logits post-forward, enabling customized behaviors.

1.2 Batching and pipelining

Serving systems exploit batching and pipelining to increase hardware utilization:

- **Static batching:** Merge requests with the same prompt length for joint prompt processing, maximizing tensor core throughput.

- **Dynamic batching:** Continuously merge ongoing decoding requests, inserting or removing sequences after each step while tracking offsets.
- **Pipeline parallelism:** Partition the model across GPUs; tokens flow through the pipeline stage by stage, reducing per-device memory footprint.

Schedulers must mitigate *batch stragglers*: requests with vastly different generation lengths reduce utilization. Typical mitigations:

1. Enforce maximum generation lengths or chunk long-form requests into smaller segments with retrieval augmentation;
2. Prioritize high-SLA traffic with weighted fair queuing or shortest-remaining-time scheduling;
3. Split warm-up (prompt) and hot (decode) phases into separated queues to tune policies independently.

1.3 Latency tuning and observability

Meeting service-level objectives (SLOs) demands optimizations across the stack:

- **Model kernels:** Fuse layer normalization, residual adds, and feed-forward operations; enable FlashAttention or fused MLP kernels to minimize memory traffic.
- **Runtime layer:** Employ asynchronous I/O, thread pools, and preallocated memory arenas to avoid allocator contention; log per-kernel latency histograms.
- **System layer:** Reduce network copies via zero-copy transports, colocate caching layers with inference pods, and isolate noisy neighbors in multi-tenant clusters.

Profilers such as NVIDIA Nsight Systems, PyTorch Profiler, or triton trace events reveal queueing delays, host-device copies, and underutilized kernels. Monitor P50/P95/P99 latencies continuously and adjust batch sizes or scheduling weights accordingly.

2 Speculative Decoding and KV Cache Reuse

2.1 Algorithm mechanics

Speculative decoding runs a lightweight draft model in parallel with the target model to reduce the number of expensive evaluations. The workflow:

1. A draft model proposes a prefix of k candidate tokens $\hat{x}_{t:t+k}$ using cached states.
2. The target model validates the proposal token by token, checking acceptance criteria and streaming accepted tokens immediately.
3. Accepted tokens reuse the draft model's KV cache; rejections trigger fallback decoding by the target model.

Figure **??** illustrates the validate-and-commit pattern in practice.

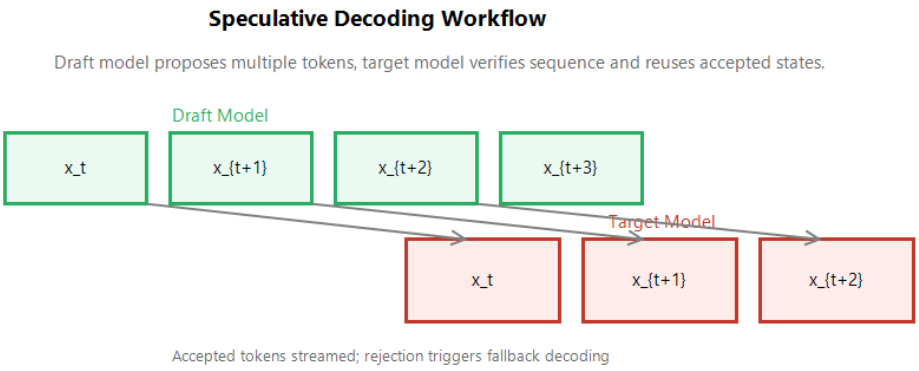

Figure 2: Speculative decoding: the draft model proposes multiple tokens, the target model verifies and reuses accepted KV states.

2.2 Acceptance rules and distribution alignment

To preserve output fidelity, the target model checks each candidate token with a rejection-sampling style test. With target distribution p_θ and draft distribution q_ϕ, the acceptance probability is

$$\alpha(x) = \min\left(1, \frac{p_\theta(x \mid x_{<t})}{q_\phi(x \mid x_{<t})}\right). \tag{1}$$

Practical considerations to maintain high acceptance:

- Distill or fine-tune the draft model from the target model to keep distributions close.
- Apply temperature scaling or sharpen the draft logits to output conservative, high-probability tokens.
- Adapt the draft length k per request based on observed acceptance rates and latency budgets.

2.3 Sharing and managing KV cache

Efficient cache sharing is crucial for realizing speedups:

- **Unified layout:** Align hidden size, head count, and tensor precision so that draft KV tensors can be copied or referenced directly by the target model.
- **Lazy writes:** Mark cache blocks as "validated" without immediate copying; perform writes only when the token is accepted to avoid rollback overhead.
- **Batch-aware scheduling:** Interleave validation across requests so the target model stays saturated even when some sequences require fallback decoding.

Well-tuned speculative decoding offers 1.5–2.5× throughput gains on popular 30B–70B models, especially when both models fit on a single GPU.

2.4 Edge cases and safeguards

- **Long-context drift:** Periodically force synchronization tokens to keep the draft model aligned when decoding beyond training lengths.
- **High-diversity sampling:** Increase acceptance slack or shorten draft spans when using high temperature or nucleus sampling, which widens the draft-target gap.
- **Graceful degradation:** Automatically disable the draft path during overload, hardware failures, or when acceptance drops below a threshold.

3 Model Quantization (4bit, 8bit, GPTQ, AWQ)

3.1 Foundations and trade-offs

Quantization reduces memory footprint and bandwidth by mapping floating-point weights and activations to low-bit integer grids. For weights $w \in \mathbb{R}^n$,

$$\tilde{w} = s \cdot \text{clip}\Big(\text{round}\Big(\frac{w}{s}\Big), q_{\min}, q_{\max}\Big), \tag{2}$$

where s is the scale and $[q_{\min}, q_{\max}]$ depends on bit width. Key trade-offs:

- **Accuracy degradation:** Lower precision introduces quantization error; calibration or quantization-aware training mitigates loss.
- **Throughput benefits:** Less memory traffic improves effective FLOPs and power efficiency.
- **Engineering complexity:** Runtime kernels must decode INT4/INT8 tensors efficiently to avoid erasing speed gains.

3.2 8-bit quantization (LLM.int8, SmoothQuant)

INT8 is a safe compromise for production workloads:

- **LLM.int8:** Splits high-variance rows to FP16 while quantizing the rest, preserving perplexity with minimal kernel changes.
- **SmoothQuant:** Jointly scales weights and activations to tame outliers, enabling fully INT8 transformer inference.

INT8 delivers roughly 2× memory savings with negligible quality loss and integrates cleanly with server-grade GPUs.

3.3 4-bit quantization and GPTQ

INT4 cuts weight storage to one quarter of FP16 while posing steeper accuracy challenges. GPTQ (Gradient Post-Training Quantization) addresses this by fitting each layer's weights:

1. Gather calibration activations to approximate the Hessian diagonal or gradient variance.
2. Quantize columns sequentially, choosing the value that minimizes reconstruction error; propagate residuals to remaining columns.
3. Allow mixed precision for sensitive columns or layers to cap perplexity regression.

GPTQ requires no gradients from the original training run and works well for offline compression but demands careful calibration data selection.

3.4 AWQ and activation-aware schemes

AWQ (Activation-aware Weight Quantization) extends INT4 by preserving critical channels:

- Estimate channel importance via activation sensitivity and allocate higher scaling factors or higher precision to important channels.
- Apply per-channel scaling in attention and MLP blocks, reducing error accumulation in long-context usage.
- Export metadata so inference engines can skip dequantization for low-impact channels.

In practice AWQ matches FP16 perplexity while retaining INT4 memory savings, and is widely supported by TensorRT-LLM, vLLM, and other runtimes.

3.5 Quantization strategy comparison

Method	Bit width	Highlights	Recommended use
LLM.int8 / SmoothQuant	8-bit	Minimal accuracy loss, channel-wise scaling	Production services requiring predictable quality
GPTQ	4-bit	Post-training, second-order column fitting	Offline or edge deployments with calibration data
AWQ	4-bit	Activation-sensitive, critical channel protection	Long-context or attention-heavy scenarios
QLoRA	4-bit weights + 16-bit adapters	Quantized base with low-rank adaptation	Memory-efficient fine-tuning and serving

Deployments often pair weight quantization with FP16 activations, selectively keeping attention output and final projection in higher precision to prevent saturation.

4 Inference Frameworks (vLLM, TGI, Exllama)

4.1 vLLM: PagedAttention and continuous batching

vLLM treats KV cache as paged virtual memory and introduces PagedAttention:

- **Fragmentation-free KV cache:** Page tables remap cache blocks to consolidate fragmented memory across concurrent sessions.
- **Continuous batching:** Dynamically insert new requests into ongoing decode loops, maximizing GPU utilization under bursty traffic.
- **Unified APIs:** Drop-in compatibility with OpenAI-style endpoints and support for HF weights, TensorRT-LLM exports, and quantized checkpoints.

Advanced features such as speculative decoding, prefix caching, and LoRA merging make vLLM a strong base for multi-tenant inference clusters.

4.2 Text Generation Inference (TGI)

HuggingFace's TGI targets production readiness end to end:

- **Optimized kernels:** Integrates FlashAttention, PagedAttention, and fused MLP kernels along with BetterTransformer fallbacks.
- **Orchestration:** Provides tensor parallelism, sharded loading, request routing, and rate limiting with Kubernetes-friendly deployment.
- **Ops tooling:** Exposes Prometheus metrics, structured logging, live config reloads, and model hot-swapping for high-availability services.

TGI pulls models directly from HuggingFace Hub, supports LoRA merging, on-demand LoRA loading, and integrates quantized artifacts without manual conversion.

4.3 Exllama: Lightweight local inference

Exllama specializes in 4-bit GPTQ models for consumer GPUs:

- **Custom kernels:** Implements high-throughput INT4 matrix multiplication and tailored cache layouts optimized for GPTQ weight packs.
- **Memory efficiency:** Stores KV cache separately and streams weights to run 30B-class models on 12GB cards.
- **Accessible tooling:** Ships with Python bindings and lightweight front-ends for local assistants, offline experimentation, and privacy-sensitive use cases.

While Exllama lacks large-scale orchestration features, it excels where deployment simplicity and hardware constraints dominate.

4.4 Choosing a serving stack

Guidance for selecting frameworks:

- Choose **vLLM** for peak throughput, speculative decoding, and multi-tenant scheduling.
- Choose **TGI** when production governance (monitoring, rate limiting, failover) is paramount.
- Choose **Exllama** for local inference, edge devices, or experimentation with 4-bit GPTQ checkpoints.
- Combine engines, e.g., run Exllama as a draft model and vLLM as the target model to unlock speculative decoding on limited hardware.

Operational recommendations

- Profile prompt, decode, and post-processing stages separately; track memory footprint alongside latency.
- Validate each optimization (speculative decoding, quantization, caching) with A/B tests capturing throughput, acceptance rates, and human evaluation scores.
- Calibrate framework-specific schedulers against traffic patterns, reserving headroom for bursty workloads and fallback paths.

Further reading

- Leviathan et al. "Fast Inference from Transformers via Speculative Decoding." ICML, 2023.
- Frantar et al. "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers." NeurIPS, 2022.
- Lin et al. "AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration." arXiv, 2023.
- Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention." arXiv, 2023.
- HuggingFace. "Text Generation Inference: Scalable Production-ready LLM Serving." Technical Report, 2024.