

Inference Engines for Large Language Models: A Comprehensive Analysis

Aryan deo

Abstract

This report provides a comprehensive analysis of inference engines for Large Language Models (LLMs), focusing on the critical aspects of model deployment and serving. We explore the fundamental concepts of inference in the context of LLM pre-training, examine the necessity of specialized inference engines, and analyze key performance parameters including quantization strategies, bandwidth considerations, and kernel optimizations. The report includes detailed comparisons of leading inference frameworks—vLLM, SGLang, NVIDIA NIM, and Triton Inference Server (TGI)—highlighting their performance characteristics, specialties, and trade-offs across different hardware configurations and batch sizes.

Contents

1	Introduction to LLM Inference	3
1.1	What is Inference in the Context of LLMs?	3
1.2	Pre-training vs. Inference	3
1.3	The Autoregressive Nature of LLM Inference	3
2	The Need for Specialized Inference Engines	4
2.1	Why Standard Deep Learning Frameworks Are Insufficient	4
2.2	Key Benefits of Inference Engines	4
2.3	Economic Impact	4
3	Key Performance Parameters	4
3.1	Quality and Quantization	4
3.1.1	Number Representation in Computing	5
3.1.2	Common Data Types	5
3.1.3	Quality vs. Size Trade-off	5
3.2	Compute Performance (TFLOPs)	5
3.2.1	Understanding FLOPs	5
3.2.2	GPU Compute Capabilities	6
3.3	Memory Bandwidth	6
3.3.1	The Bandwidth Bottleneck	6
3.3.2	Compute-Bound vs. Memory-Bound Operations	6
3.3.3	Example Calculation	7
3.4	Quantization and Speed Trade-offs	7

3.4.1	The Quantization Paradox	7
3.4.2	Solution: Hardware-Accelerated Quantization	7
3.5	Kernel Optimization	7
3.5.1	What are Kernels?	7
3.5.2	Fused Kernels	8
3.6	GPU Utilization	8
4	Inference Engine Comparison	8
4.1	Tested Frameworks	8
4.2	Test Configuration	8
4.3	Performance Results	9
4.3.1	Batch Size 1 (Single Request)	9
4.3.2	Batch Size 64 (Concurrent Requests)	10
4.4	Framework Specialties and Differences	10
4.4.1	vLLM	10
4.4.2	SGLang	11
4.4.3	NVIDIA NIM	11
4.4.4	TGI (Triton Inference Server)	11
4.5	Feature Comparison Matrix	12
5	Cost Analysis Across GPU Types	12
5.1	Hardware Configurations Tested	12
5.2	Model Size Considerations	12
5.3	Recommendations by Model Size	12
5.3.1	Llama 3.1 8B	12
5.3.2	Llama 3.1 70B	13
5.3.3	Llama 3.1 405B	13
5.4	Cost Per Million Tokens	13
6	Optimal Inference Configuration Recommendations	14
6.1	Recommended Configurations	14
6.2	Decision Framework	14
6.3	Framework Selection Guide	14
7	Advanced Topics	15
7.1	Continuous Batching	15
7.2	KV Cache Management	15
7.3	Speculative Decoding	15
8	Conclusion	16
8.1	Key Takeaways	16
8.2	Future Directions	16
8.3	Final Recommendations	16

1 Introduction to LLM Inference

1.1 What is Inference in the Context of LLMs?

Inference is the process of using a trained Large Language Model to generate predictions or outputs based on new input data. Unlike the pre-training phase, where the model learns patterns from massive datasets through gradient descent and backpropagation, inference involves:

- **Forward pass only:** No gradient computation or weight updates
- **Autoregressive generation:** Tokens are generated sequentially, one at a time
- **Real-time constraints:** Users expect low-latency responses
- **High throughput requirements:** Serving multiple concurrent requests

1.2 Pre-training vs. Inference

The key differences between pre-training and inference are summarized in Table 1.

Table 1: Pre-training vs. Inference Comparison		
Aspect	Pre-training	Inference
Computation	Forward + Backward pass	Forward pass only
Memory	Stores gradients & optimizer states	Stores model weights only
Batch size	Very large (thousands)	Variable (1-64+)
Latency	Not critical	Critical for user experience
Duration	Days to weeks	Milliseconds to seconds
Hardware	Multi-GPU clusters	Single or few GPUs
Cost metric	Total training cost	Cost per token

1.3 The Autoregressive Nature of LLM Inference

LLMs generate text token-by-token in an autoregressive manner:

$$P(x_1, x_2, \dots, x_n) = \prod_{i=1}^n P(x_i | x_1, x_2, \dots, x_{i-1}) \quad (1)$$

This sequential generation creates unique challenges:

- Each token depends on all previous tokens
- Cannot parallelize across the sequence dimension
- Memory grows with sequence length (KV cache)
- Bandwidth becomes a critical bottleneck

2 The Need for Specialized Inference Engines

2.1 Why Standard Deep Learning Frameworks Are Insufficient

Traditional frameworks like PyTorch and TensorFlow were designed primarily for training, not inference. They lack:

1. **Efficient memory management:** No built-in KV cache optimization
2. **Batching strategies:** Cannot efficiently batch requests of varying lengths
3. **Quantization support:** Limited low-precision inference capabilities
4. **Hardware optimization:** Not optimized for inference-specific kernels
5. **Request scheduling:** No sophisticated queuing and priority systems

2.2 Key Benefits of Inference Engines

Specialized inference engines provide:

- **Higher throughput:** 5-10x improvement through continuous batching
- **Lower latency:** Optimized kernels and memory access patterns
- **Better resource utilization:** Efficient GPU memory management
- **Cost reduction:** Serve more requests with fewer GPUs
- **Production features:** Request queuing, monitoring, API compatibility

2.3 Economic Impact

For a production deployment serving 1 million requests per day:

- **Without optimization:** May require 8 H100 GPUs @ \$30,000/month = \$240,000/month
- **With inference engine:** May require only 4 H100 GPUs = \$120,000/month
- **Annual savings:** \$1.44 million

3 Key Performance Parameters

3.1 Quality and Quantization

Quantization reduces the precision of model weights to decrease memory usage and increase speed. The quality depends on the number format used.

3.1.1 Number Representation in Computing

A floating-point number in binary format consists of:

$$\text{Number} = (-1)^{\text{sign}} \times \text{mantissa} \times 2^{\text{exponent}} \quad (2)$$

For example, the number 15 can be represented as:

$$\text{Decimal: } 1.5 \times 10^1 \quad (3)$$

$$\text{Binary: } 1.111_2 \times 2^{11_2} = 1.111_2 \times 2^3 \quad (4)$$

where the mantissa is $1.111_2 = 1.875$ and the exponent is 3.

3.1.2 Common Data Types

Table 2 summarizes common data types used in LLM inference.

Table 2: Data Types for LLM Inference				
Datatype	Sign Bits	Mantissa Bits	Exponent Bits	Quality
FLOAT32	1	23	8	Excellent
BFLOAT16	1	7	8	Excellent
FLOAT16	1	10	5	Very good
FP8-E5M2	1	2	5	Very good
NF4	0	4	0	Good
FP4	1	1	2	Good?
INT4	0	4	0	OK

3.1.3 Quality vs. Size Trade-off

- **FLOAT32/BFLOAT16:** Maintain near-perfect quality, suitable for production
- **FP8:** Minimal quality degradation, 2x memory reduction
- **INT4/NF4:** Noticeable but acceptable quality loss, 4x memory reduction

3.2 Compute Performance (TFLOPs)

The theoretical compute capability varies significantly across data types and hardware.

3.2.1 Understanding FLOPs

FLOP (Floating Point Operation) is a single arithmetic operation (addition or multiplication). **TFLOPs** (Tera FLOPs) represents trillions of operations per second.

For LLM inference:

$$\text{Compute Required} \approx 2 \times N_{\text{params}} \times N_{\text{tokens}} \quad (5)$$

where N_{params} is the number of model parameters.

3.2.2 GPU Compute Capabilities

Table 3 shows compute capabilities across different GPUs and data types.

Table 3: GPU Compute Capabilities (TFLOPs)

GPU Type	FLOAT32	BFLOAT16	FP8	Bandwidth (TB/s)
T4 (Colab)	8.1	-	-	0.3
A6000	39	150	-	0.7
A40	37	150	-	0.7
A100	19.5	312	-	2.0
H100	67	1,000	2,000	3.3
B100	60	1,750	3,500	-

Key Insight: Lower-precision formats can provide 10-30x speedup on modern GPUs (H100, B100).

3.3 Memory Bandwidth

3.3.1 The Bandwidth Bottleneck

In LLM inference, model weights are stored in VRAM (Video RAM) but computations occur in SRAM (on-chip cache). The bandwidth between these two determines the maximum speed.

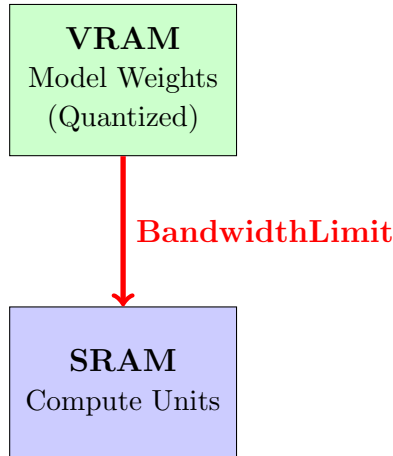


Figure 1: Memory Hierarchy in GPU

3.3.2 Compute-Bound vs. Memory-Bound Operations

- **Compute-bound:** Speed limited by TFLOPs (rare in inference)
- **Memory-bound:** Speed limited by bandwidth (common in inference)

For a typical inference scenario:

$$\text{Time per token} = \frac{\text{Model size (bytes)}}{\text{Bandwidth (bytes/s)}} \quad (6)$$

3.3.3 Example Calculation

For Llama 3.1 8B model in FP16 (2 bytes per parameter):

$$\text{Model size} = 8 \times 10^9 \times 2 = 16 \text{ GB} \quad (7)$$

$$\text{On A100 (2 TB/s)} : \frac{16 \text{ GB}}{2000 \text{ GB/s}} = 8 \text{ ms per token} \quad (8)$$

$$\text{On H100 (3.3 TB/s)} : \frac{16 \text{ GB}}{3300 \text{ GB/s}} \approx 5 \text{ ms per token} \quad (9)$$

3.4 Quantization and Speed Trade-offs

3.4.1 The Quantization Paradox

Quantization to 4-bit reduces memory footprint by 4x, but:

- If GPU cannot compute natively in 4-bit
- Must dequantize to 16-bit before computation
- Dequantization overhead can negate bandwidth savings

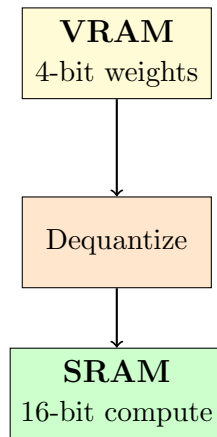


Figure 2: Dequantization Overhead

3.4.2 Solution: Hardware-Accelerated Quantization

Modern GPUs support native low-precision compute:

- **H100/B100**: Native FP8 and FP4 support
- **Speedup**: 2-4x faster than FP16 without quality loss

3.5 Kernel Optimization

3.5.1 What are Kernels?

Kernels are low-level GPU functions that execute operations. Standard kernels may:

- Dequantize data
- Perform matrix multiplication
- Execute these as separate operations

3.5.2 Fused Kernels

Fused kernels combine multiple operations into one, reducing memory transfers:

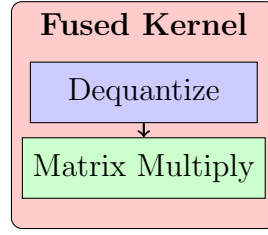


Figure 3: Fused Kernel Architecture

Popular Optimized Kernels:

- **Marlin:** INT4 quantized inference
- **AWQ:** Activation-aware weight quantization
- **Neural Magic:** Sparsity and quantization

3.6 GPU Utilization

Effective kernels maximize GPU utilization through:

- **Efficient scheduling:** Keep all compute units busy
- **Memory coalescing:** Optimize memory access patterns
- **Reduced synchronization:** Minimize idle time

4 Inference Engine Comparison

4.1 Tested Frameworks

This analysis compares four leading inference engines:

1. **vLLM:** Popular open-source inference engine
2. **SGLang:** High-performance engine with advanced scheduling
3. **NVIDIA NIM:** Official NVIDIA inference microservice
4. **TGI (Triton Inference Server):** Hugging Face’s production server

4.2 Test Configuration

Model: Llama 3.1 8B (FP8 quantization)

Hardware: 1x H100 SXM

Prompt: "Write a long essay on the topic of spring."

Max tokens: 500

Batch sizes: 1 (single request) and 64 (concurrent requests)

4.3 Performance Results

4.3.1 Batch Size 1 (Single Request)

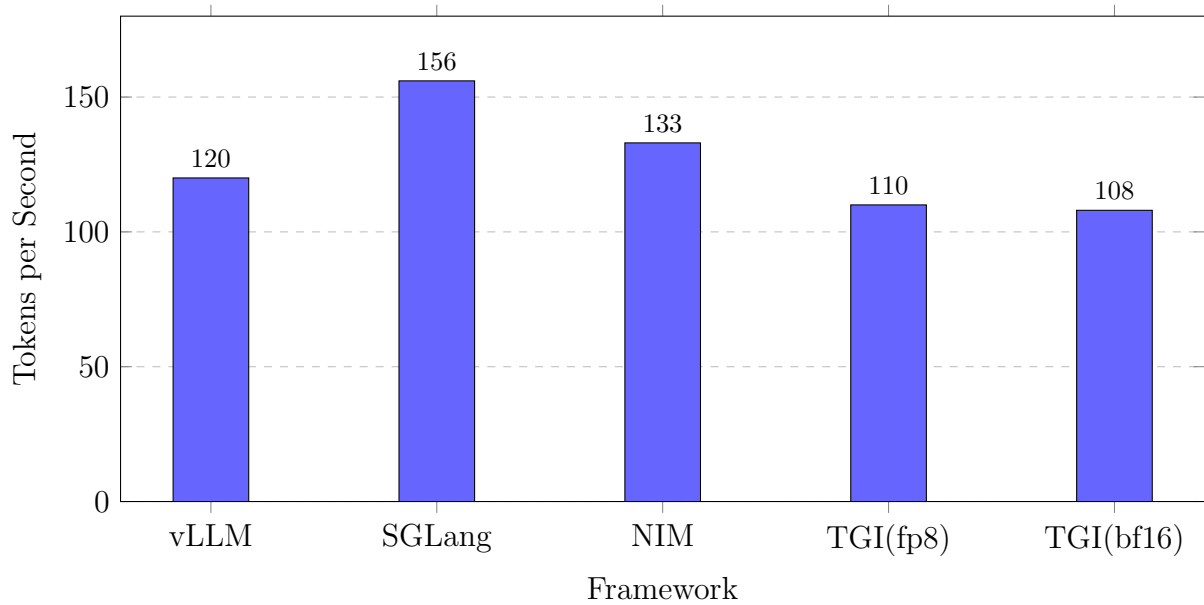


Figure 4: Throughput Comparison at Batch Size 1

Key Findings:

- SGLang achieves highest throughput (156 tok/s)
- 30% faster than vLLM
- 17% faster than NVIDIA NIM
- TGI shows similar performance between FP8 and BF16

4.3.2 Batch Size 64 (Concurrent Requests)

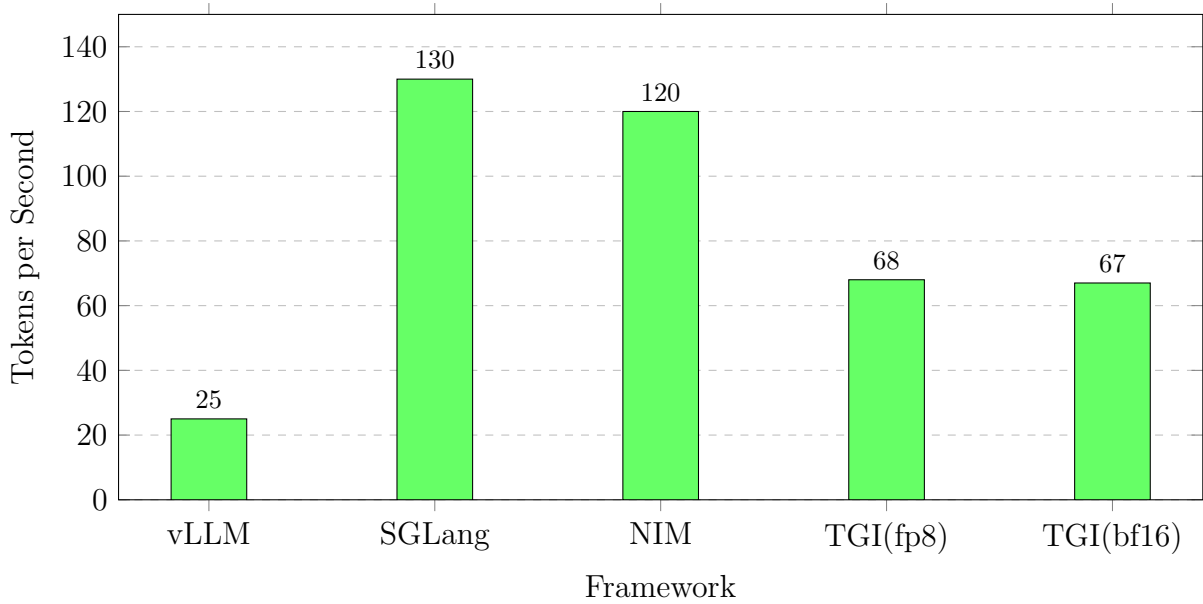


Figure 5: Throughput Comparison at Batch Size 64

Key Findings:

- SGLang maintains high performance (130 tok/s)
- vLLM shows significant degradation (25 tok/s - 5x slower)
- NVIDIA NIM performs well (120 tok/s)
- TGI shows moderate performance

4.4 Framework Specialties and Differences

4.4.1 vLLM

Strengths:

- Excellent single-request performance
- Wide model support
- Active community and documentation
- PagedAttention for efficient memory management

Weaknesses:

- Poor performance at large batch sizes
- Less optimized scheduling

Best for: Low-concurrency applications, development, prototyping

4.4.2 SGLang

Strengths:

- Superior performance across all batch sizes
- Advanced request scheduling (RadixAttention)
- Efficient KV cache management
- Prefix caching for repeated prompts

Weaknesses:

- Newer project with smaller community
- Fewer production deployments

Best for: High-throughput production deployments, batch processing

4.4.3 NVIDIA NIM

Strengths:

- Official NVIDIA support
- Optimized for NVIDIA GPUs
- Good batch performance
- Production-ready packaging

Weaknesses:

- Closed-source
- NVIDIA hardware only
- Less flexibility than open-source alternatives

Best for: Enterprise deployments with NVIDIA infrastructure

4.4.4 TGI (Triton Inference Server)

Strengths:

- Hugging Face integration
- Supports multiple frameworks
- Good ecosystem compatibility
- Structured generation support

Weaknesses:

- Lower throughput than SGLang/NIM
- FP8 doesn't show significant advantage over BF16

Best for: Hugging Face-centric workflows, multi-framework deployments

4.5 Feature Comparison Matrix

Table 4: Framework Feature Comparison

Feature	vLLM	SGLang	NIM	TGI
Single Request Perf.	Good	Excellent	Good	Moderate
Batch Performance	Poor	Excellent	Good	Moderate
Structured Generation	Yes	Yes	Yes	Yes
Quantization Support	FP8, INT4	FP8, INT4	FP8, INT4	FP8, INT4
Open Source	Yes	Yes	No	Yes
Production Ready	Yes	Emerging	Yes	Yes
Community Size	Large	Growing	N/A	Large
HF Integration	Good	Moderate	Moderate	Excellent

5 Cost Analysis Across GPU Types

5.1 Hardware Configurations Tested

The analysis examined cost-effectiveness across different GPU types:

- **A40:** 48GB VRAM, 0.7 TB/s bandwidth
- **A6000:** 48GB VRAM, 0.7 TB/s bandwidth
- **A100:** 80GB VRAM, 2.0 TB/s bandwidth
- **H100 SXM:** 80GB VRAM, 3.3 TB/s bandwidth

5.2 Model Size Considerations

Table 5: Model Sizes and GPU Requirements

Model	FP8 Size	INT4 Size	Min GPUs (FP8)	Min GPUs (INT4)
Llama 8B	~8 GB	~4 GB	1x A40	1x A40
Llama 70B	~70 GB	~35 GB	2x A40/1x A100	1x A40
Llama 405B	~405 GB	~203 GB	8x H100	4x H100

5.3 Recommendations by Model Size

5.3.1 Llama 3.1 8B

High Quality (FP8):

- **Recommended:** 1x A40 with SGLang
- **Cost:** \$1-2/hour (\$30-60/month)
- **Throughput:** ~130 tok/s (batch 64)

Budget Option (INT4):

- 1x A40 with INT4 quantization
- Marginal throughput improvement
- Not recommended (quality degradation not worth minor speedup)

5.3.2 Llama 3.1 70B

High Quality (FP8):

- **Recommended:** 4x A40 or 2x H100 with SGLang
- **Cost:** 4x A40: \$4-8/hour; 2x H100: \$6-12/hour
- **Alternative:** 2x A100 for balance of cost/performance

Budget Option (INT4):

- 2x A40 with INT4
- Lower hourly cost but higher cost per token (slower throughput)
- Consider for low-concurrency scenarios

5.3.3 Llama 3.1 405B

High Quality (FP8):

- **Recommended:** 8x H100 with SGLang
- **Cost:** \$24-48/hour
- **Best performance:** Maximum throughput

Budget Option (INT4):

- 4x H100 with INT4
- **Cost:** \$12-24/hour
- Trade-off: Lower hourly cost, higher cost per token

5.4 Cost Per Million Tokens

The actual cost-effectiveness depends on throughput achieved:

$$\text{Cost per M tokens} = \frac{\text{Hourly GPU cost}}{\text{Throughput (tokens/s)} \times 3600} \quad (10)$$

Example for Llama 70B:

- 4x A40 @ \$6/hour, 50 tok/s: \$33/M tokens
- 2x A100 @ \$8/hour, 80 tok/s: \$27/M tokens
- 2x H100 @ \$10/hour, 120 tok/s: \$23/M tokens

6 Optimal Inference Configuration Recommendations

6.1 Recommended Configurations

Table 6: Optimal GPU and Quantization Recommendations

GPU Type	Quality Priority	Speed Priority	Best Use Case
T4/Colab	FLOAT16 (quality)	GGUF (speed)	4/5-bit Development, testing
Mac M1	FLOAT16 (quality)	GGUF (speed)	4/5-bit Local development
A6000/A40	FP8-Marlin (quality)	INT4-Marlin (speed)	Production (medium scale)
A100	FP8-Marlin (quality)	INT4-Marlin (speed)	Production (balanced)
H100	FP8-Marlin (quality)	INT4-Marlin (speed)	Production (high throughput)
B100	FP8/FP4 native	FP4 native (speed)	Next-gen deployment

6.2 Decision Framework

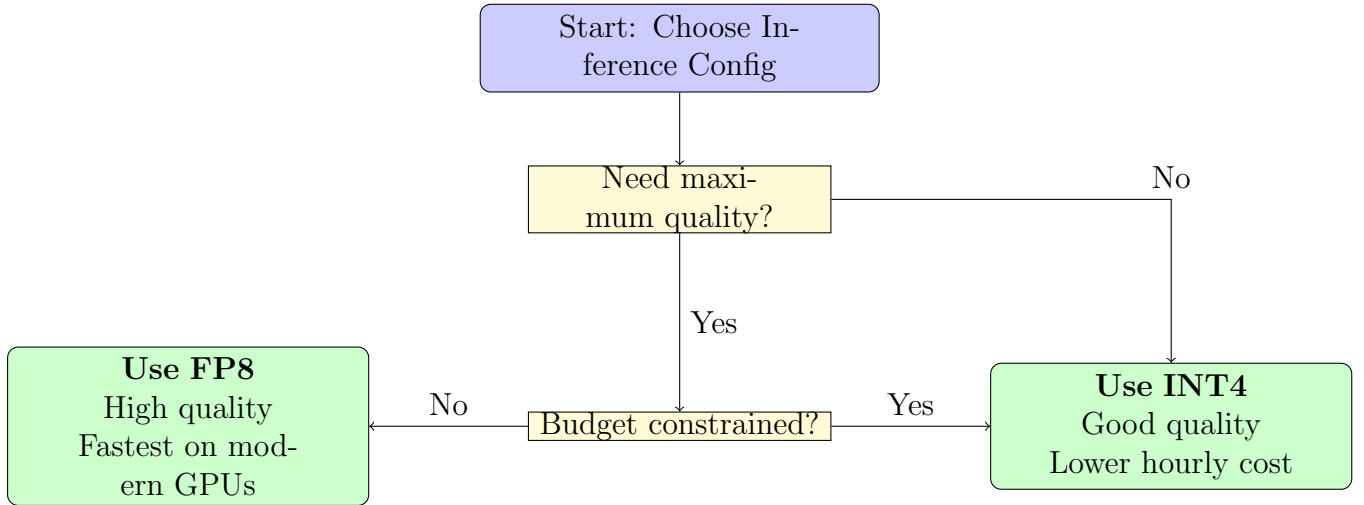


Figure 6: Quantization Decision Framework

6.3 Framework Selection Guide

1. Choose SGLang if:

- High concurrency expected (batch size > 10)
- Maximum throughput required
- Can tolerate newer, less proven technology

2. Choose vLLM if:

- Low concurrency (batch size < 10)
 - Need extensive documentation and community support
 - Rapid prototyping and development
3. **Choose NVIDIA NIM if:**
- Enterprise deployment with NVIDIA GPUs
 - Need official vendor support
 - Good balance of performance needed
4. **Choose TGI if:**
- Heavy Hugging Face ecosystem integration
 - Need multi-framework support
 - Moderate performance acceptable

7 Advanced Topics

7.1 Continuous Batching

Modern inference engines use continuous batching to maximize GPU utilization:

- **Traditional batching:** Wait for batch to fill before processing
- **Continuous batching:** Add/remove requests dynamically
- **Benefit:** Higher throughput, lower average latency

7.2 KV Cache Management

The Key-Value cache stores previous token representations:

$$\text{KV cache size} = 2 \times n_{\text{layers}} \times n_{\text{heads}} \times d_{\text{head}} \times \text{seq_len} \times \text{batch_size} \quad (11)$$

Efficient management strategies:

- **PagedAttention (vLLM):** Manage KV cache in pages like OS memory
- **RadixAttention (SGLang):** Share KV cache across requests with common prefixes

7.3 Speculative Decoding

Use a smaller "draft" model to generate multiple tokens, then verify with the main model:

- Can achieve 2-3x speedup
- No quality degradation
- Effective for models with high token acceptance rate

8 Conclusion

8.1 Key Takeaways

1. **Inference engines are essential** for production LLM deployments, offering 5-10x improvements over naive implementations
2. **SGLang leads in throughput**, especially for high-concurrency scenarios, while vLLM excels in single-request latency
3. **FP8 quantization** provides the best balance of quality and speed on modern GPUs (H100, A100)
4. **INT4 quantization** reduces hourly costs but may increase cost-per-token due to lower throughput
5. **Hardware choice matters**: H100 offers best performance but A100/A40 can be more cost-effective depending on workload
6. **Batch size significantly impacts** framework performance—test with realistic workloads

8.2 Future Directions

The field of LLM inference continues to evolve rapidly:

- **FP4 native support**: Next-generation GPUs (B100) with native FP4 compute
- **Mixture of Experts (MoE)**: Specialized inference techniques for MoE models
- **Multi-modal inference**: Optimizations for vision-language models
- **Edge deployment**: Inference on mobile and embedded devices

8.3 Final Recommendations

For most production deployments:

- Start with **SGLang + FP8** on **A100 or H100** GPUs
- Use **vLLM** for development and low-concurrency applications
- Monitor cost-per-token metrics, not just hourly costs
- Test with realistic batch sizes and workload patterns
- Consider structured generation support for JSON/API outputs