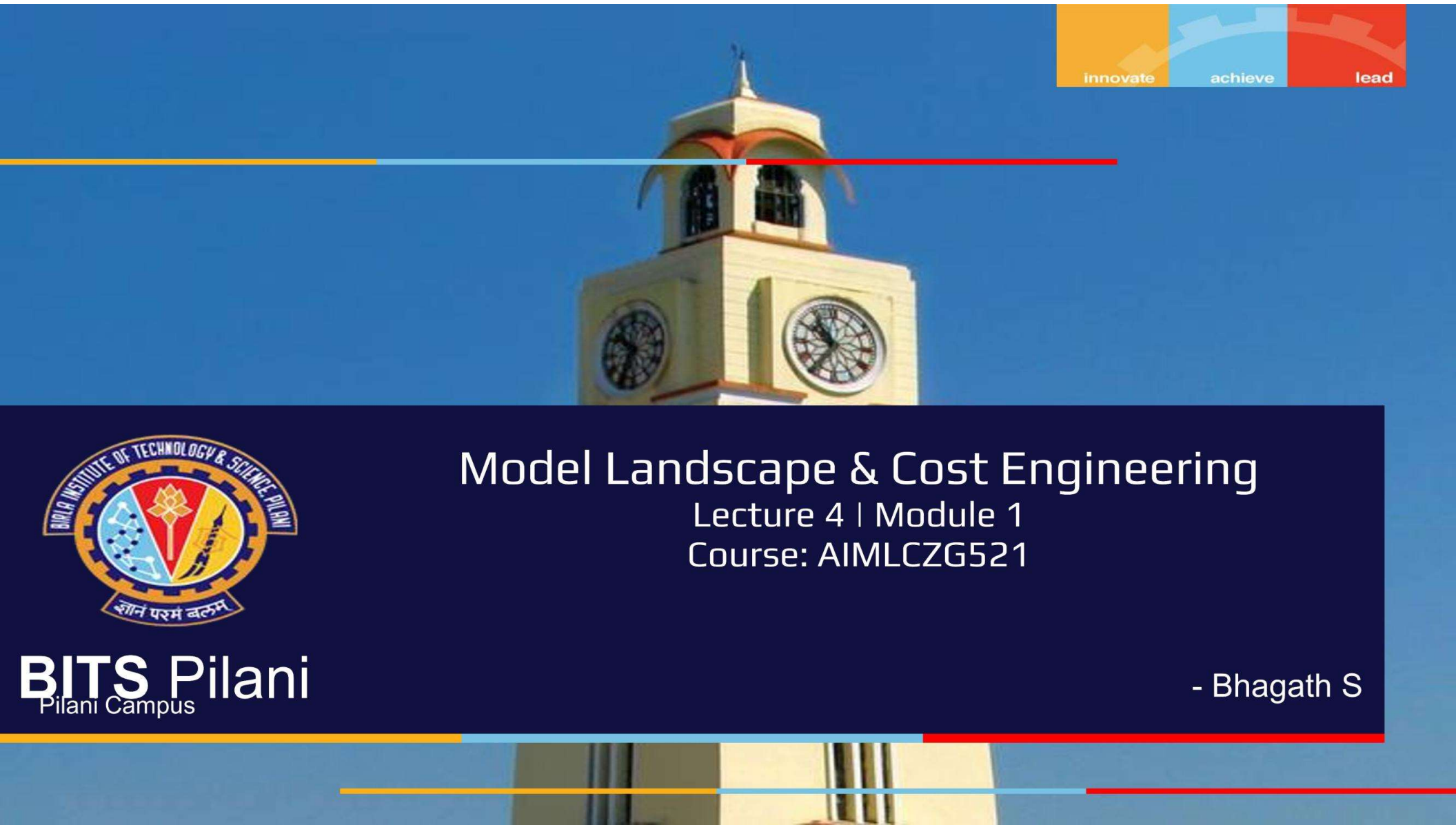


innovate achieve lead

# Model Landscape & Cost Engineering

Lecture 4 | Module 1

Course: AIMLCZG521

BITS Pilani

Pilani Campus

- Bhagath S

# Learning Objectives

- Navigate the current LLM ecosystem and understand different model architectures
- Master quantization techniques for efficient inference and deployment
- Calculate GPU memory requirements for different model configurations
- Design cost-effective production systems with optimal model selection
- Implement practical cost optimization strategies for enterprise deployments

# Agenda


## Part 1: Model Landscape

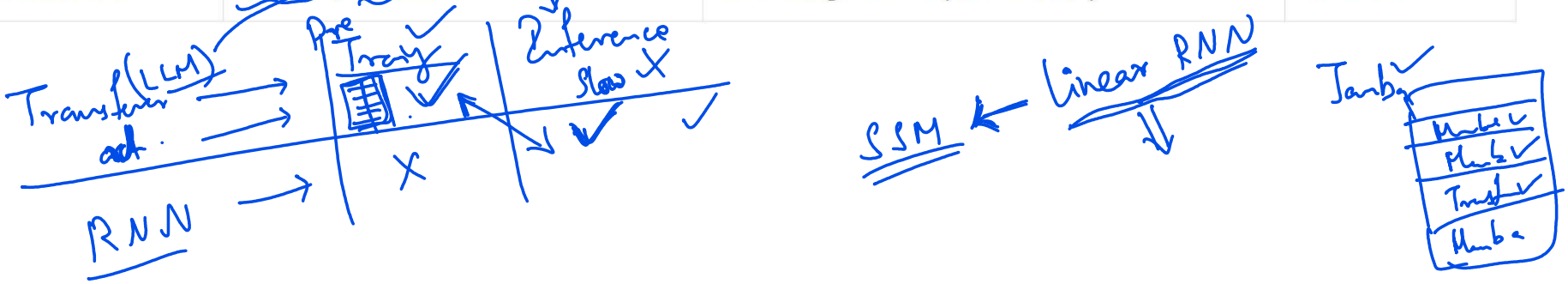
- Transformer-based Models
- Mixture-of-Experts (MoE)
- Small Language Models
- State Space Models

## Part 2: Cost Engineering

- Quantization Techniques
- GPU Memory Estimation
- Token-based Pricing
- Optimization Strategies

# The 2025 LLM Landscape

| Architecture Type | Key Models | Best Use Case | Cost Profile |
|---|---|---|---|
| **Dense Transformers** | GPT-4, Claude 3.5, Llama 3.3 | General purpose, complex reasoning | High |
| **Mixture-of-Experts** | Mixtral 8x7B, DeepSeek-V2 | Cost-effective at scale | Medium |
| **Small LMs (1-3B)** | Phi-3, Gemma, Qwen | Specialized tasks, edge devices | Low |
| **State Space Models** | Mamba, Jamba | Ultra-long context (1M+ tokens) | Variable |

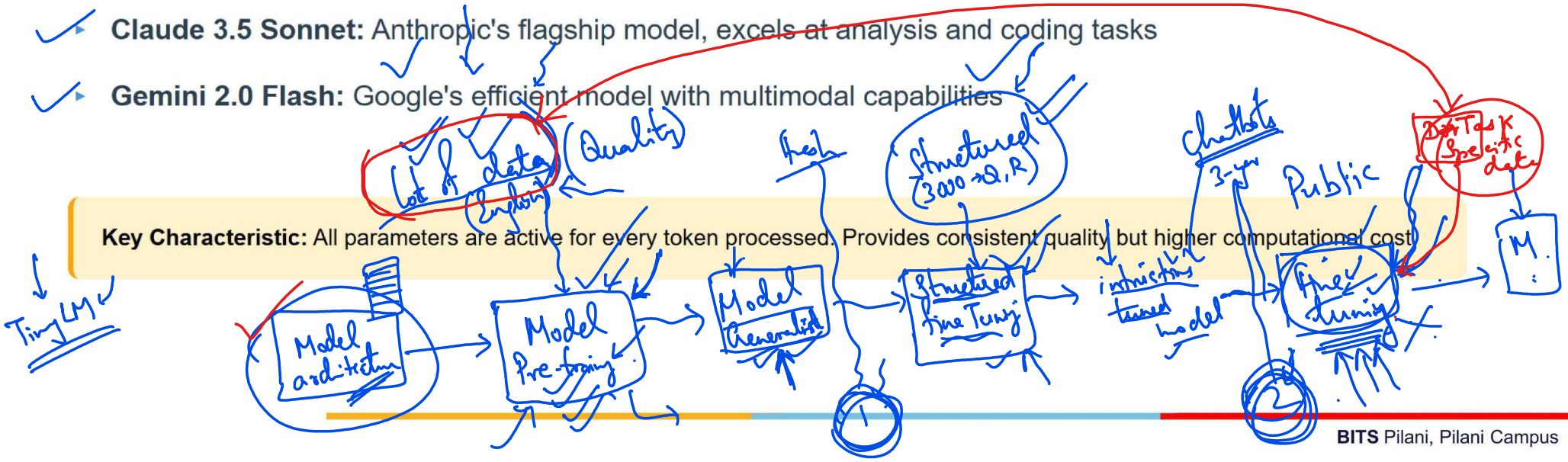


# Dense Transformer Models

## State-of-the-Art Foundation Models (2025)

- **Llama 3.3 70B:** Meta's latest open-weight model, competitive with GPT-4 on many benchmarks
- **GPT-4 Turbo & GPT-4o:** OpenAI's multimodal models with 128K context window
- **Claude 3.5 Sonnet:** Anthropic's flagship model, excels at analysis and coding tasks
- **Gemini 2.0 Flash:** Google's efficient model with multimodal capabilities

**Key Characteristic:** All parameters are active for every token processed. Provides consistent quality but higher computational cost



# Mixture of Experts: Router Mechanism

## How Sparse Activation Achieves Efficiency

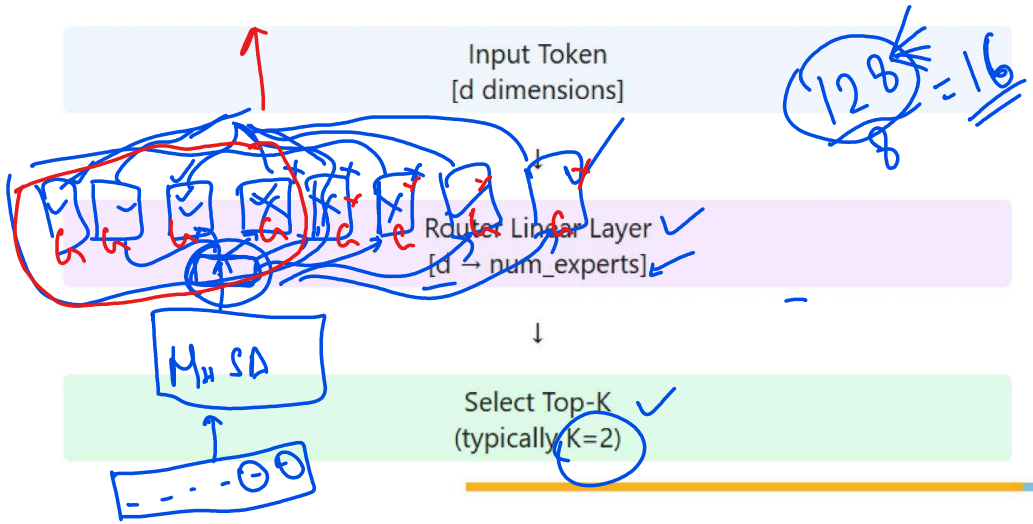
### The Routing Decision

```
router_logits = W_router · x
top_k_experts = TopK(router_logits, k=2)
weights = softmax(top_k_logits)
output = Σ (w_i × Expert_i(x))
```

### Example: Mixtral 8x7B

- 8 expert networks (each 7B parameters)
- Router selects top-2 experts per token
- Total params: 47B, Active per token: 13B
- Efficiency: 47B / 13B ≈ 72% parameter reduction in computation

### Router Network




### Load Balancing

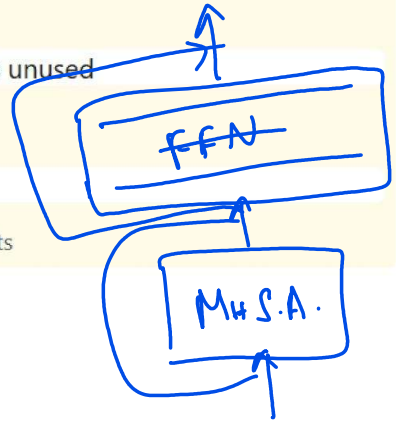
**Challenge: Expert Collapse**

Router may favor certain experts, leaving others unused

**Solution: Auxiliary Loss**

$L_{aux} = \alpha \times load\_balance\_loss$

Encourages uniform distribution of tokens across experts



# Mixture of Experts Models

## Cost-Effective Scaling Strategy

- Mixtral 8x7B: 47B total params, only 13B active per token
- DeepSeek-V2: 236B total, 21B active - extremely cost-efficient
- DBRX: 132B total, 36B active parameters

Router Network: Selects which expert(s) to use for each token, enabling sparse activation

### Efficiency Formula

Cost Reduction = 1 - (Active Params / Total Params)

Mixtral: 1 - (13B/47B) ≈ 72% reduction

**Key Challenges:**

- **Load Balancing:** Experts may become imbalanced - some overused, others underutilized
- **Inference Latency:** Router overhead adds 5-15% latency
- **Memory Requirements:** All experts must be loaded in memory

# Small Language Models (SLMs)

## Specialized, Efficient Models (1-3B Parameters)

| Model | Parameters | Strengths | Deployment |
|---|---|---|---|
| Phi-3 Mini | 3.8B | Reasoning, coding, math | Mobile, edge devices |
| Gemma 2B | 2B | General purpose, safety-tuned | On-device inference |
| Qwen 1.5-1.8B | 1.8B | Multilingual, fast inference | Resource-constrained envs |

**Production Tip:** Use SLMs for classification, extraction, and routing tasks. Reserve larger models for complex reasoning and generation.

# Attention Mechanism Fundamentals

Understanding the Core of Transformer Architecture

## Self-Attention in Transformers

**Query (Q):** What am I looking for?

**Key (K):** What do I contain?

**Value (V):** What information do I have?

### Attention Formula

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k})\,V$$

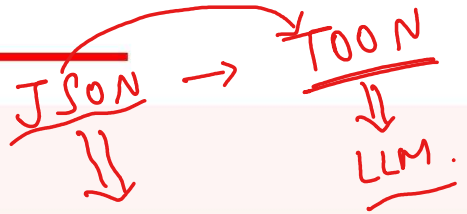
**Step 1: Score**
Compute $QK^T$

**Step 2: Scale**
Divide by $\sqrt{d_k}$

**Step 3: Weight**
Softmax then multiply V

# Transformer's Quadratic Complexity Problem

Why Attention is Expensive for Long Sequences




## The Complexity Challenge

$$O(n^2 \cdot d)$$

n = sequence length, d = model dimension

### Memory Requirements

- Sequence = 512 tokens → $512^2$ = 262K attention scores
- Sequence = 2048 tokens → $2048^2$ = 4.2M scores
- Sequence = 8192 tokens → $8192^2$ = 67M scores

Grows quadratically!

### Why $O(n^2)$?

**Every token** attends to **every other token**

$QK^T$ = [n × d] · [d × n]
Result = [n × n] matrix

For each of n layers, we compute $n^2$ attention scores

This quadratic scaling makes processing long documents (100K+ tokens) impractical

# State Space Models: Linear Complexity Solution

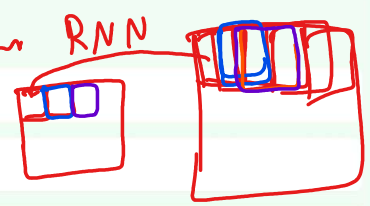
## How SSMs Address the Quadratic Complexity Problem

### ❌ Transformer Attention

- Complexity: $O(n^2 \cdot d)$
- Each token sees all tokens
- High memory for long sequences
- Excellent at random access

### ✓ State Space Models

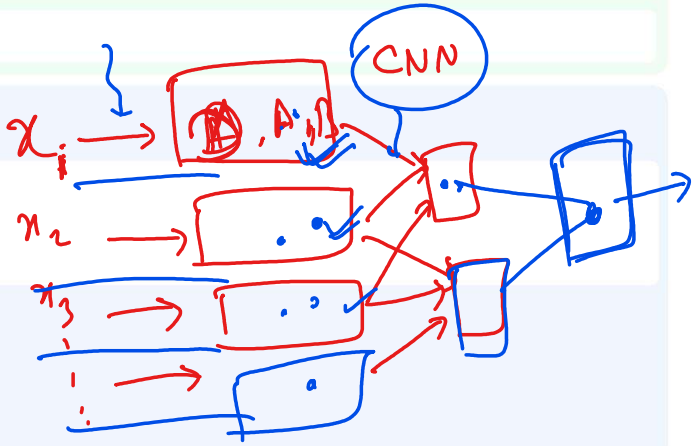
- Complexity: $O(n \cdot d)$
- Sequential state propagation
- Constant memory per step
- Linear time scaling




### State Space Representation

$$h_t = A \cdot h_{t-1} + B \cdot x_t$$

$$y_t = C \cdot h_t + D \cdot x_t$$




- $h_t$: Hidden state (compressed representation of history)
- $x_t$: Current input token
- $y_t$: Output for current position

🎯 **Key Insight: Process sequentially, maintain fixed-size state → Linear complexity!**

# State Space Models (SSMs)

## Linear Complexity for Ultra-Long Context

### Why SSMs?

- **Transformer:** Quadratic complexity - expensive for long sequences
- **SSM:** Linear complexity using state space representations
- **Mamba:** Can handle 1M+ token contexts efficiently

**Complexity Comparison**

Transformer: Quadratic time

Mamba/SSM: Linear time

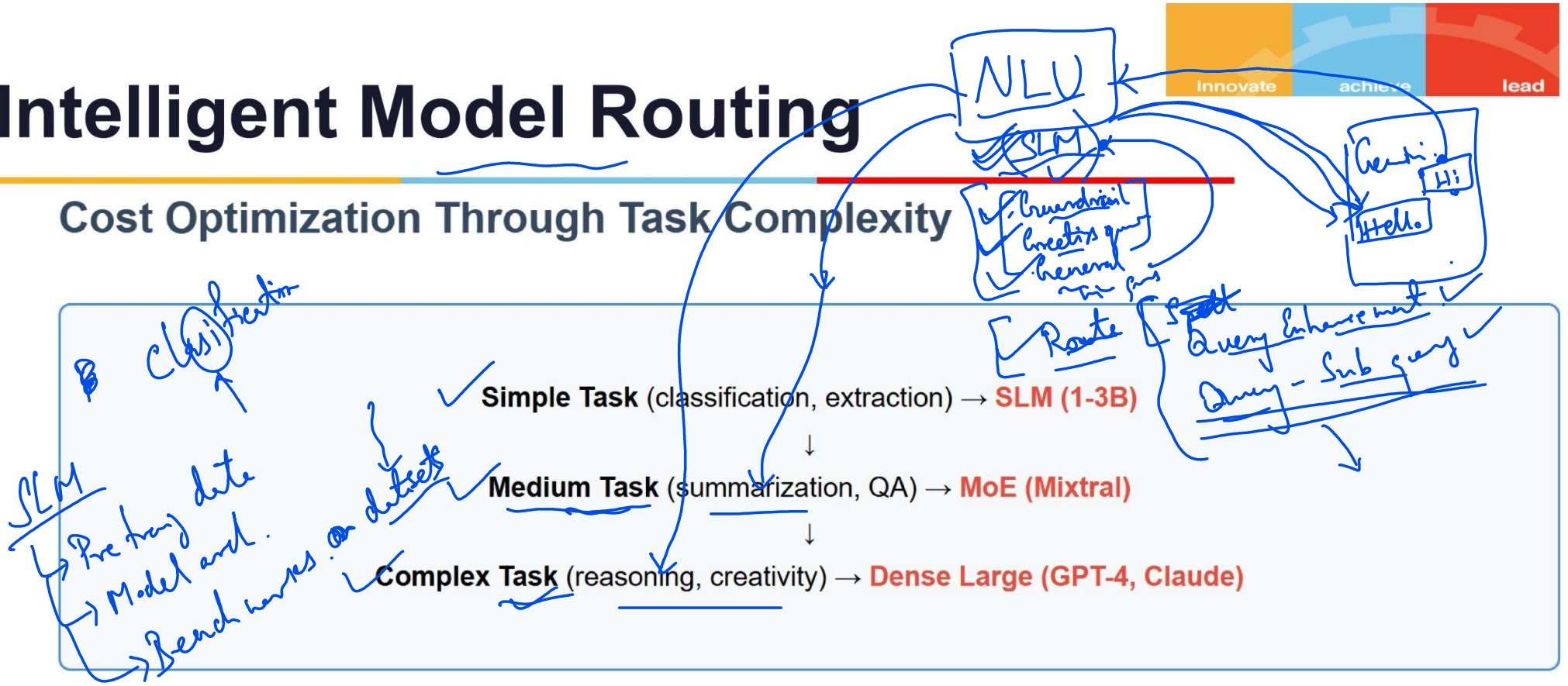

**Use Case:** Processing entire codebases, long documents, or extended conversations without chunking

**Limitations:**

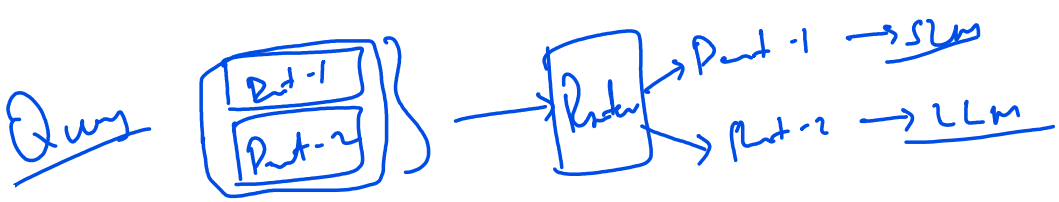
- **In-Context Learning:** Weaker than transformers at random access tasks
- **Practical Limits:** Memory constraints limit to ~1M tokens currently
- **Production Maturity:** Limited adoption vs transformers
- **Hybrid Architectures:** Jamba combines Mamba with attention layers

# Intelligent Model Routing

## Cost Optimization Through Task Complexity

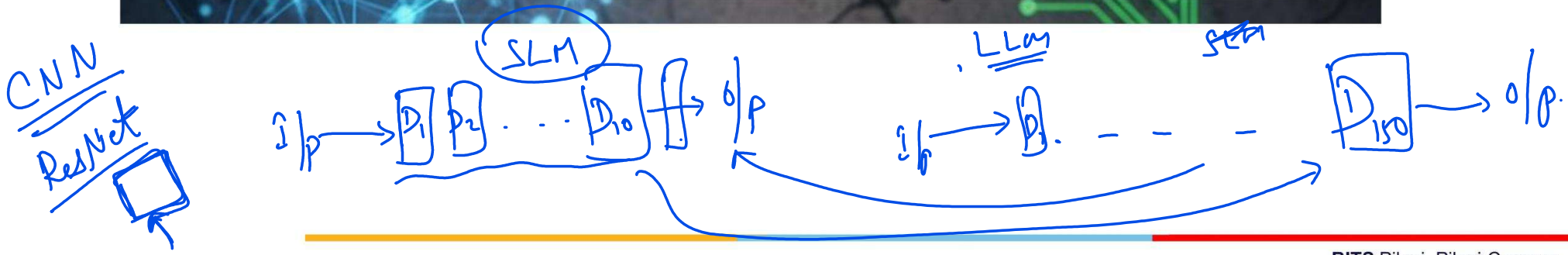



**Real-World Example:** Customer support chatbot uses Phi-3 for intent classification (95% of queries), routes complex issues to GPT-4 (5% of queries). Result: 70% cost reduction with maintained quality.






We've seen how models differ in architecture and scaling. Now, let's see how these choices impact cost.



# Quantization: Making Models Efficient

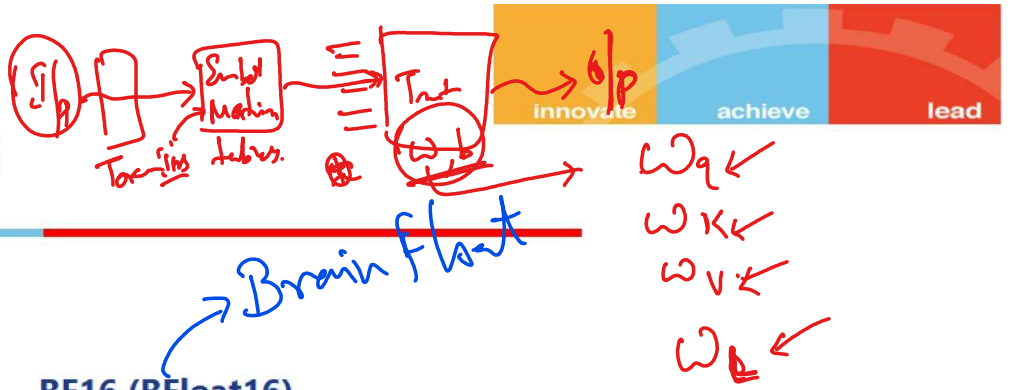
innovate achieve lead

## Reducing Precision Without Sacrificing Quality

| Precision | Bits/Param | Memory (7B) | Quality | Speed | Best Use |
|---|---|---|---|---|---|
| FP32 | 32 bits | 28 GB | 100% | 1x | Training |
| FP16 | 16 bits | 14 GB | ~99.9% | 1.5-2x | Training/Inference |
| BF16 | 16 bits | 14 GB | ~99.9% | 1.5-2x | LLM Inference |
| INT8 | 8 bits | 7 GB | ~99% | 2-4x | Production |
| NF4 | 4 bits | 3.5 GB | ~98% | 3-6x | Edge/Cost |

Production Tip: INT8 provides optimal balance - 4x memory reduction, 2-4x speed, with less than 1% quality loss

# Floating Point Formats




## Understanding Precision Trade-offs

### FP32 (Float32)

| Sign:1 | Exp:8 | Mantissa:23 |
|---|---|---|

- Standard for training
- Wide dynamic range

### FP16 (Float16)

| Sign:1 | Exp:5 | Mantissa:10 |
|---|---|---|

- Can cause underflow/overflow

### BF16 (BFloat16)

| Sign:1 | Exp:8 | Mantissa:7 |
|---|---|---|

- Same range as FP32
- Preferred for LLM inference

### FP8 (Emerging 2025)

**E4M3 (Training)**
4-bit exponent, 3-bit mantissa

**E5M2 (Inference)**
5-bit exponent, 2-bit mantissa

🎯 **BF16 is the sweet spot - 50% memory reduction with minimal accuracy loss**

# Post-Training Quantization (PTQ)


## Quantize Pre-trained Models Without Retraining

### GPTQ

- Layer-wise quantization
- Minimizes reconstruction error
- 4-bit with ~1% accuracy loss
- Good for GPUs

### AWQ (Activation-aware)

- Protects important weights
- Based on activation patterns
- Better quality at 4-bit
- Slightly slower than GPTQ

# Low-Rank Adaptation (LoRA)

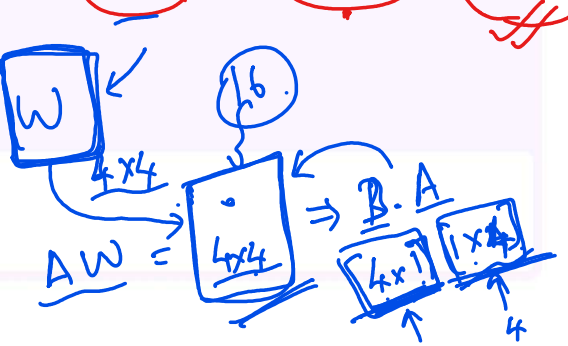
## Efficient Fine-Tuning Through Parameter Decomposition


### The Core Concept

Instead of updating all model parameters, inject small trainable matrices alongside frozen weights

$$W_{new} = W_{frozen} + \Delta W$$

$$\text{where } \Delta W = B \cdot A \text{ (low-rank decomposition)}$$




**Traditional Fine-Tuning**

Pre-trained Model
(7B parameters)

↓ Update ALL ↓

Fine-tuned Model
(7B parameters)

❌ 7B × 4 bytes ≈ 28 GB storage per model

**LoRA Fine-Tuning**

Pre-trained Model (Frozen)
(7B parameters)

Matrix A (d×r) | Matrix B (r×d)

LoRA adapters
(8-16M params)

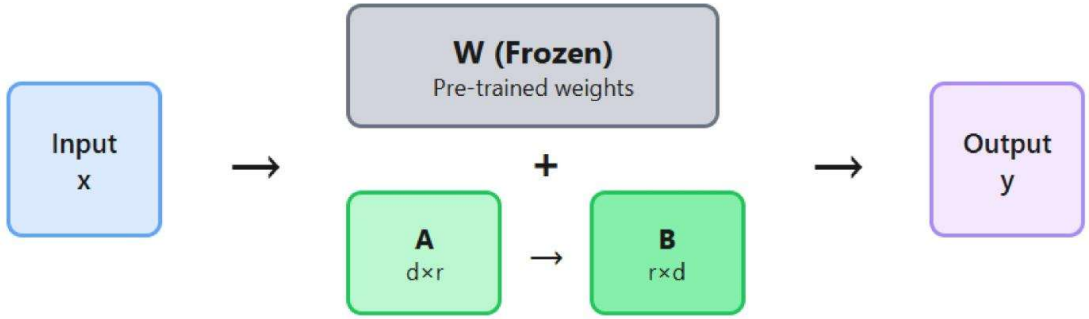
✓ Only 32-64 MB per adaptation!

### Why "Low-Rank"?

Instead of updating a d×d matrix (e.g., 4096×4096), we use two smaller matrices: A (4096×8) and B (8×4096). Rank r=8 is much smaller than d=4096, hence "low-rank".

# LoRA Architecture Visualization

## How LoRA Adapters Integrate with Pre-trained Models




### Frozen Path

- Original model weights
- No gradient updates
- Preserves pre-training

### LoRA Path

- Trainable adapters
- Gradient updates only here
- Task-specific learning

### Combined Output

- Sum both paths
- Best of both worlds
- Efficient adaptation

### Key Parameters:

- **r (rank):** Typically 4-64, controls adapter size
- **α (scaling):** Controls LoRA contribution
- **Target layers:** Usually attention weights (Q, V)
- **Memory:** <1% of full fine-tuning

# QLoRA: Quantization + Fine-Tuning


## Efficient Parameter-Efficient Training

- **Base Model:** Quantized to 4-bit (NF4) - frozen during training
- **LoRA Adapters:** Small trainable matrices in FP16/BF16
- **Memory Savings:** 70B model trainable on single 48GB GPU
- **Performance:** Matches full fine-tuning quality in most cases

**Memory Calculation**

Full Fine-Tuning: 70B × 4 bytes × 4 (optimizer states) ≈ 1120 GB
QLoRA: 70B × 0.5 bytes + LoRA (1GB) ≈ 36 GB

31× Memory Reduction!

# Quantization Decision Guide

| Scenario | Recommended Precision | Rationale |
|---|---|---|
| Training from scratch | FP32 or BF16 | Stability during gradient updates |
| Fine-tuning large models | QLoRA (4-bit + LoRA) | Memory efficiency, good quality |
| Production inference (cloud) | BF16 or INT8 | Balance of speed and quality |
| Edge deployment | INT8 or NF4 | Minimal memory footprint |
| Real-time applications | INT8 | Fastest inference speed |

**Rule of Thumb:** For most production use cases, 4-bit quantization (GPTQ/AWQ) is sufficient. Only use FP16/BF16 when quality is absolutely critical.

# GPU Memory Estimation

## Understanding Memory Components

**Total Memory Formula**

$$M_{total} = M_{weights} + M_{KV\text{-}cache} + M_{activation}$$

- **Model Weights:** Fixed size based on parameters and precision
- **KV-Cache:** Grows with sequence length and batch size
- **Activations:** Temporary memory during forward pass
- **Rule of Thumb:** Total ≈ 1.2× model size for inference

**Example:** A 7B model in BF16 (14GB) needs approximately 17GB VRAM for inference with reasonable batch sizes.

# Why Do We Need KV-Cache?

innovate achieve lead

## Understanding Attention Computation in Autoregressive Generation

### The Inefficiency Problem

During text generation, transformers generate one token at a time. Each new token needs to attend to ALL previous tokens.

### Without KV-Cache (Naive Approach)

```
Token 1: "The" → Compute Q₁, K₁, V₁
Token 2: "cat" → Compute Q₁, K₁, V₁, Q₂, K₂, V₂ (recompute Token 1!)
Token 3: "is" → Compute Q₁, K₁, V₁, Q₂, K₂, V₂, Q₃, K₃, V₃ (recompute all!)
Token n: Recompute everything for tokens 1 to n-1
```

❌ Computational cost grows as $O(n^2)$ for n tokens!

### With KV-Cache (Optimized)

```
Token 1: "The" → Compute K₁, V₁ → Cache them
Token 2: "cat" → Reuse K₁, V₁, compute K₂, V₂ → Cache
Token 3: "is" → Reuse K₁, V₁, K₂, V₂, compute K₃, V₃ → Cache
Token n: Reuse all cached K, V, compute only new K_n, V_n
```

✓ Only compute new token's K, V → O(n) for n tokens!

### Why Only Cache K and V?

- **Query (Q):** Changes with each new token position
- **Keys (K) & Values (V):** Represent previous context, remain constant
- We only need to compute Q for the new token and use cached K, V for attention

# KV-Cache: Technical Implementation

## What Exactly is Stored and How Memory Grows

### What's in the Cache?

**For each layer:** Store K and V matrices

**For each attention head:** Separate K, V

**For each token:** One vector per head

```
Cache size per token =
2 × layers × heads × d_head × precision
```

### Memory Growth Pattern

**Prefill Phase:** Process input prompt → Build initial cache

**Generation Phase:** Add one entry per generated token

**Memory = Linear in sequence length**

Grows with: batch_size × seq_length

### Attention Computation with Cache

```
# New token arrives at position t
Q_new = W_Q × x_t   # Compute query for new token

K_cached = [K_1, K_2, ..., K_{t-1}]   # Retrieve from cache
V_cached = [V_1, V_2, ..., V_{t-1}]   # Retrieve from cache

K_t = W_K × x_t   # Compute key for new token
V_t = W_V × x_t   # Compute value for new token

K_all = concat(K_cached, K_t)   # Append to cache
V_all = concat(V_cached, V_t)   # Append to cache

attention = softmax(Q_new × K_all^T / √d_k) × V_all
```

### Trade-off:

✓ Speedup: 10-100x faster generation (no recomputation)

✗ Memory: Can consume more memory than model weights for long contexts!

# KV-Cache: The Hidden Memory Consumer

innovate achieve lead

## Understanding and Optimizing Key-Value Cache

### KV-Cache Memory Formula

```
MKV = 2 × layers × heads × d_head × seq_len × batch × 2 bytes
```

**Example: Llama 3.3 8B**

- Layers: 32
- Heads: 32
- d_head: 128
- Precision: BF16 (2 bytes)

**For 2K context, batch=1:**

MKV = 2 × 32 × 32 × 128 × 2048 × 1 × 2

**≈ 1 GB**

# Hands-On: Memory Calculation

## Calculate VRAM for Claude 3.5 Sonnet Inference

### Given Configuration:

- Parameters: 175B
- Precision: BF16 (2 bytes)
- Sequence Length: 4096 tokens
- Batch Size: 4
- Layers: 96, Heads: 96, d_head: 128

### 1. Weights

175B × 2 bytes

**= 350 GB**

### 2. KV-Cache

2 × 96 × 96 × 128 × 4096 × 4 × 2

**≈ 96 GB**

### 3. Activations

batch × seq × hidden × layers × 20

**≈ 38 GB**

### Total VRAM Required

**350 GB + 96 GB + 38 GB ≈ 484 GB**

💡 **Practical:** Requires 6-7× A100 40GB or 2× H100 80GB with tensor parallelism

# Memory-Efficient Inference at Scale

## Advanced Techniques for Production Deployment

### FlashAttention-2

- Reduces memory from quadratic to linear
- 2-4x speed improvement
- 10-20x memory reduction
- Essential for 8K+ contexts

### PagedAttention

- Virtual memory for KV-cache
- Reduces waste from 60% to under 10%
- 2-5x higher throughput
- Better GPU utilization

### CPU Offloading

- KV-cache to CPU memory
- 10x cheaper memory
- Trade: 2-3x slower
- Handles 10x longer contexts

### Parallelism Strategies

**Tensor Parallelism (TP)**

- Split layers across GPUs
- Lower latency
- Best for 2-8 GPUs

**Pipeline Parallelism (PP)**

- Split depth-wise
- Higher throughput
- Best for 8+ GPUs

🎯 **Recommendation:** FlashAttention-2 + PagedAttention as baseline. Add TP for 70B+ models.

# Token Economics: Understanding API Costs

## 2025 Pricing Landscape (per 1M tokens)

| Model | Input | Output | Cached |
|---|---|---|---|
| GPT-4 Turbo | $10 | $30 | $5 |
| Claude 3.5 Sonnet | $3 | $15 | $0.30 |
| Gemini 2.0 Flash | $0.075 | $0.30 | $0.019 |
| Mixtral 8x7B | $0.60 | $0.60 | N/A |
| Self-Hosted (vLLM) | $0.10-0.30* | $0.10-0.30* | N/A |

**Key Observation:**

Output tokens cost 3-10× more than input. Minimize generation length and maximize caching.

⚠️ **Hidden Costs:** APIs include auto-scaling and zero maintenance. Self-hosting requires DevOps ($10K/month).

Costs mentioned in the slides are tentative. Refer the respective documentations for the actual current costs.


# Prompt Caching: 90% Cost Reduction

## Reuse Repeated Context Across Requests

### How It Works

- Cache static prompts (system, examples)
- 5-minute TTL (Anthropic), 1 hour (OpenAI)
- Minimum 1024 tokens for Claude
- 90% discount on cached tokens

### Cost Example

**Without Caching:**

10K × 100 requests = 1M × $3 = $3

**With Caching:**

First: 10K × $3 = $0.03

Next 99: 10K × $0.30 × 99 = $0.297

**Total: $0.327 (89% savings!)**

### Optimization Strategies

**1. Prompt Structure**

- Static content first
- Dynamic queries last

**2. Cache Invalidation**

- Any change = cache miss
- Use fixed templates

# Production Cost Optimization Playbook

### Immediate Actions

- Enable prompt caching for repeated context
- Use smaller models for simple tasks
- Batch non-urgent requests
- Set max_tokens appropriately

### Advanced Techniques

- Model routing based on complexity
- Self-hosted quantized models
- Speculative decoding (2x faster)
- Efficient retrieval (reduce context)

**Real-World Impact:** A production RAG system reduced costs from $12K/month to $2K/month using: 50% prompt caching + router (Gemma → Claude) + 4K max context instead of 16K.

# Emerging Trends in LLM Optimization (2025)

## What's Next in Model Efficiency

### 1-Bit LLMs (BitNet)

- Weights in -101
- 10-20x memory reduction
- 3-5x energy efficiency
- Competitive quality

**Status:**
Research stage, production by late 2025

### Mixture of Depths

- Dynamic computation per layer
- Tokens skip expensive layers
- 40-50% FLOPs reduction
- Combines with MoE

### FlashAttention-3

- Hardware-specific (H100, H200)
- 3-5x faster than FA-2
- 1M+ context support
- Reduced memory footprint

### Hardware-Aware Quant

- Optimized per GPU architecture
- FP8 on H100, INT4 on A100
- 2-3x better utilization
- Auto format selection

# Self-Hosting vs. API: Break-Even Analysis

## Making the Right Infrastructure Decision

| Factor | API | Self-Hosted |
|---|---|---|
| Upfront Cost | $0 | $5K-50K |
| Monthly (1B tokens) | $3,000 | $1,000 |
| Maintenance | None | $10K/month |
| Scalability | Instant | Hours-days |
| Data Privacy | Shared infra | Full control |
| Latency | Variable | Predictable |

Costs mentioned in the slides are tentative. Refer the respective documentations for the actual current costs.

# Key Takeaways

- **Model Selection Matters:** Use the smallest model that achieves your quality bar. SLMs for 80% of tasks, large models for 20%.
- **Quantization is Essential:** 4-bit quantization (GPTQ/AWQ) provides 4× memory reduction with <2% quality loss.
- **KV-Cache Dominates Memory:** For long contexts, KV-cache can exceed model weights. Plan accordingly.
- **Prompt Caching = Free Money:** 90% cost reduction on repeated context. Always enable for production systems.
- **Monitor Everything:** Track token usage, costs per model, and cache hit rates. Optimize based on data.
- **Start with APIs:** Self-hosting only makes sense at >10B tokens/month scale.

# References & Resources


- Dettmers et al. (2023). "QLoRA: Efficient Finetuning of Quantized LLMs." arXiv:2305.14314
- Lin et al. (2023). "AWQ: Activation-aware Weight Quantization for LLM Compression." arXiv:2306.00978
- Ma & Shen (2024). "The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits." arXiv:2402.17764
- Gu & Dao (2023). "Mamba: Linear-Time Sequence Modeling with Selective State Spaces." arXiv:2312.00752
- Anthropic Prompt Caching: https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching
- OpenAI Prompt Caching: https://platform.openai.com/docs/guides/prompt-caching