1. How to optimize the cost of an overall LLM System?

Cost optimization requires a multi-layered approach:

- **Model Right-Sizing:** Don't use GPT-4 for everything. Use a "Router" architecture. Route simple queries (greeting, classification) to cheaper/faster models (Llama-3-8B) and complex queries to SOTA models.
- **Caching (Semantic Cache):** Store the embedding of user queries. If a new query is semantically similar (cosine similarity > 0.9) to a previous one, serve the cached answer immediately. This avoids GPU inference entirely.
- **Prompt Engineering:** Optimize prompts to be concise. Fewer input tokens = lower cost. Use "Reference-based" instructions rather than pasting massive context if not needed.
- **Batching:** In self-hosted scenarios, use **Continuous Batching** to saturate the GPU. Running a GPU at 20% utilization costs the same electricity as 100%, so maximize throughput.
- **Quantization:** Serve models in 4-bit or 8-bit. This allows you to fit larger models on cheaper, consumer-grade GPUs rather than renting expensive A100s.

---

2. What are Mixture of Expert (MoE) models?

Mixture of Experts (MoE) is a transformer architecture designed to scale model size (parameters) without scaling inference cost linearly.

- **Concept:** Instead of one massive dense network (where every token passes through every parameter), the model is divided into smaller "experts" (sub-networks).
- **Gating Network:** For every token, a "Router" or "Gate" decides which expert is best suited to process it.
  - *Example:* If the token is "Javascript", it might route to the "Coding Expert."
- **Sparse Activation:** In a model with 8 experts, typically only **top-2** are activated per token.
- **Benefit:** You get the knowledge capacity of a massive model (e.g., Mixtral 8x7B has 47B params) but the inference speed/cost of a much smaller model (it only uses ~13B active params per token).

---

3. How to build a production-grade RAG system?

A production RAG system has five distinct stages, each with specific optimizations:

1. **Ingestion Pipeline:**
   - **Parsing:** Use specialized parsers (like Unstructured or LlamaParse) to handle tables and PDFs correctly.
   - **Advanced Chunking:** Use Semantic Chunking or Parent-Child Chunking to preserve context.
2. **Indexing:**
   - **Vector DB:** Use HNSW index for speed.
   - **Hybrid Index:** Create a parallel Keyword Index (BM25) to catch acronyms and exact IDs.

3. **Retrieval:**
    - **Hybrid Search:** Query both indexes and merge results (RRF).
    - **Metadata Filtering:** Apply date/category filters *before* semantic search.
4. **Refinement (The "Secret Sauce"):**
    - **Re-Ranking:** Use a Cross-Encoder (e.g., Cohere Rerank) to re-score the top 50 retrieved chunks and pick the top 5 most relevant. This is the single biggest booster of accuracy.
5. **Generation:**
    - **Prompting:** Use "Chain of Verification" to reduce hallucinations.
    - **Citations:** Force the model to cite the chunk ID used for the answer.

---

4. What is the FP8 variable and what are its advantages?

FP8 (Floating Point 8-bit) is a data format that uses only 8 bits to represent a number, compared to 16 bits (FP16/BF16) or 32 bits (FP32).

- **Two Formats:**
    - **E4M3:** (4 Exponent bits, 3 Mantissa bits). Higher precision, lower dynamic range. Used for **Weights**.
    - **E5M2:** (5 Exponent bits, 2 Mantissa bits). Lower precision, higher dynamic range. Used for **Gradients/Activations** (matches BF16 dynamic range).
- **Advantages:**
    - **Memory:** Halves the VRAM usage compared to FP16.
    - **Bandwidth:** Doubles the effective memory bandwidth (data moves from HBM to Tensor Cores 2x faster).
    - **Compute:** NVIDIA H100 Tensor Cores can perform matrix multiplication in FP8 at **2x the speed** of FP16.

---

5. How to train LLM with low precision without compromising accuracy?

Training in FP8 or INT8 is risky because small gradients can underflow to zero (vanish). To fix this:

1. **Mixed Precision:** Do the heavy Matrix Multiplications in low precision (FP8) for speed, but perform the **Accumulation** (summing up the results) in high precision (FP32) to preserve small details.
2. **Scaling Factors:** Since FP8 has a limited range, multiply the tensors by a "Scaling Factor" to shift values into the representable range before quantization, then un-scale them after.
3. **Stochastic Rounding:** Instead of rounding to the nearest number (which biases towards zero), randomly round up or down based on probability. This preserves the statistical "average" of the gradients over millions of operations.

---

6. How to calculate the size of KV cache?

During inference, the model must remember the Key and Value matrices for every token generated so far.

Formula (Per Request):

$$\text{Size}_{\text{KV}} = 2 \times n_{\text{layers}} \times n_{\text{heads}} \times d_{\text{head}} \times \text{seq\_len} \times \text{Precision (Bytes)}$$

- **2:** Because we store Keys and Values.
- **Precision:** 2 bytes for FP16.

Simplified Formula:

$$\text{Size}_{\text{KV}} = 2 \times n_{\text{layers}} \times d_{\text{model}} \times \text{seq\_len} \times 2$$

- Example (Llama-2-7B): 32 layers, 4096 hidden dim. For a context of 1000 tokens: $2 \times 32 \times 4096 \times 1000 \times 2 \approx 524 \text{MB}$ per user.

---

7. Explain dimension of each layer in Multi-Headed Attention

Let:

- $B$: Batch Size
- $S$: Sequence Length
- $D$: Model Dimension (e.g., 512)
- $h$: Number of heads (e.g., 8)
- $d_k$: Head Dimension ($D / h = 64$)

The Matrices:

1. **Input ($X$):** Shape $(B, S, D)$.
2. **Projections ($W_q, W_k, W_v$):**
   - These are linear layers mapping $D \rightarrow D$.
   - Shape: $(D, D)$.
   - *Note:* Internally, they split the output into $h$ heads, so effectively they project to $(D, h \times d_k)$.
3. **Q, K, V Vectors:**
   - After projection and splitting heads: $(B, h, S, d_k)$.
4. **Attention Scores ($Q \times K^T$):**
   - Dot product results in a square matrix: $(B, h, S, S)$.
5. **Output Projection ($W_o$):**
   - After concatenating heads back together: $(B, S, D)$.
   - Final projection: $(D, D)$.

---

8. How do you make sure that the attention layer focuses on the right part of the input?

You don't manually "make" it focus; you **train** it to minimize loss. However, the mechanism works due to the **Dot Product**:

- **Mechanism:** Attention is essentially a similarity search. The Dot Product $Q \cdot K$ measures alignment. If the vector for "Subject" aligns with the vector for "Verb", the score is high.
- **Training Objective:** During backpropagation, if the model predicts the wrong next token, the gradient signal flows back through the Attention Weights.
    - If the model ignored the word "not" and predicted "happy" instead of "sad," the gradients will adjust the W_q and W_k matrices to ensure that next time, the "Query" for the current word pays higher "Attention" to the "Key" of the word "not."
- **Positional Encoding:** This ensures the model focuses on the right *relative* part (e.g., looking at the word immediately preceding the current one).