## **Section 1 – Conceptual Understanding (Short Answer)**

### **1. Explain the main difference between model training and model inference.**

**Model training** is the process of teaching a model to learn patterns from large amounts of data by adjusting its internal parameters (weights and biases) through optimization algorithms like gradient descent. Training is computationally intensive, requires GPUs/TPUs, and can take hours or days depending on the model size.  
 In contrast, **model inference** refers to the stage where the trained model is used to make predictions on new, unseen data. It’s a *forward pass only* process — no parameter updates happen here.

In simple terms:

* **Training = Learning phase (compute-heavy, infrequent)**
* **Inference = Prediction phase (latency-sensitive, frequent)**

In LLMs, training might happen once over petabytes of text, but inference happens millions of times per day — so optimization during inference is crucial for real-world applications.

### **2. What techniques can you use to reduce inference latency for large language models (LLMs)?**

Reducing inference latency involves both **software-level** and **hardware-level** optimizations. Some of the key techniques include:

* **Quantization:** Convert high-precision (FP32) weights to lower precision (FP16, INT8, or INT4) to reduce memory bandwidth and computation time.
* **Model pruning and distillation:** Remove redundant neurons or layers or distill a smaller model that mimics a larger one’s behavior.
* **Batching requests:** Combine multiple user requests into a single forward pass to maximize GPU utilization.
* **Speculative decoding:** Use a smaller draft model to generate several tokens ahead and verify them with the main model, improving token generation throughput.
* **Caching and KV-cache reuse:** Store intermediate attention states (key-value pairs) so each new token generation doesn’t recompute everything.
* **Optimized kernels:** Use frameworks like TensorRT, ONNX Runtime, or DeepSpeed-Inference for GPU-optimized execution.
* **Hardware acceleration:** Use GPUs, TPUs, or inference accelerators (like AWS Inferentia or NVIDIA Tensor Cores).

Together, these optimizations can bring down latency from hundreds of milliseconds to under 100ms per request.

### **3. Describe how quantization and speculative decoding improve inference performance.**

* **Quantization:**  
   Quantization reduces the numeric precision of model parameters (e.g., from FP32 → INT8).  
   This decreases model size, reduces memory transfer time, and speeds up matrix multiplications — all with minimal loss in accuracy. For instance, quantizing a 13B LLM can reduce its memory footprint by 60–70% and inference latency by 2–3×.  
   Frameworks like PyTorch’s torch.quantization, Hugging Face’s bitsandbytes, or ggml for LLaMA are popular for this.
* **Speculative Decoding:**  
   Instead of generating tokens one by one with a large model, speculative decoding uses a **smaller “draft” model** to predict multiple tokens at once.  
   The large model then **verifies** those tokens in parallel.  
   If they’re correct, they’re accepted; if not, the main model corrects them.  
   This approach can yield up to 2×–3× faster decoding without quality loss and is used in production systems like OpenAI’s GPT models.

In essence — **quantization** accelerates the *math*, while **speculative decoding** accelerates the *generation process*.

### **4. What are some challenges of deploying open-source LLMs (e.g., LLaMA or Qwen) in production?**

Deploying open-source LLMs in production is powerful but challenging due to several operational and technical reasons:

1. **Hardware requirements:** Running large models (7B–70B parameters) demands multiple high-memory GPUs or tensor parallelism, which is costly and complex to manage on-premise.
2. **Latency vs. cost trade-off:** Scaling for low latency (e.g., <100ms) often means keeping multiple replicas always on — increasing costs.
3. **Optimization complexity:** You must manually handle quantization, tensor parallelism, batching, and KV caching for performance tuning.
4. **Security and compliance:** Open-source models don’t come with managed safety filters or audit logs like OpenAI’s models.
5. **Maintenance overhead:** Frequent model updates, bug fixes, and fine-tuning pipelines require expertise and continuous monitoring.
6. **Model versioning & rollback:** Handling multiple versions across nodes and ensuring reproducibility is non-trivial.

Essentially, with open-source LLMs, **you gain control but lose convenience** — everything from scaling to safety has to be built in-house.

### **5. Explain how an inference API like Baseten or Modal differs from using a hosted API like OpenAI.**

**Hosted APIs (e.g., OpenAI, Anthropic):**

* The provider manages *everything* — infrastructure, scaling, optimization, and updates.
* You only send requests and receive responses via REST API.
* Pros: Zero DevOps burden, high reliability.
* Cons: No control over model weights, limited customization, higher per-request cost.

**Inference APIs (e.g., Baseten, Modal, Replicate):**

* These platforms let you deploy *your own models* (like LLaMA or Falcon) in a managed inference environment.
* They handle containerization, GPU scheduling, auto-scaling, and endpoint exposure.
* You can bring custom models, quantization, or fine-tuning logic while still avoiding full infrastructure management.

In short:

* **OpenAI = “Use our model”** (SaaS)
* **Baseten/Modal = “Host your model easily”** (PaaS)

This gives you flexibility between full self-hosting and fully managed APIs.