**Section – 1 Conceptual Understanding**

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

Training is the learning stage—where the model analyzes large datasets, adjusts internal weights, and finds relationships in data to minimize prediction errors. It’s resource-heavy and typically done in dedicated environments using GPUs or cloud training nodes.

**Inference**, by contrast, happens after this learning phase. It’s the part where the trained model is deployed to make real-world predictions or generate outputs. In practice, this is the stage users actually interact with—for instance, when a chatbot answers a prompt, or a recommendation system suggests a product.

From an operations perspective, training defines the model’s abilities, while inference measures how efficiently those abilities are used in real scenarios.

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

In fast-paced applications like customer support or interactive chat systems, **latency** the delay between input and model response can make or break user experience. Improving inference latency typically involves a mix of algorithmic and infrastructure-level optimizations.

Teams often use techniques such as:

* **Model compression**, using quantization or pruning to reduce compute load.
* **Knowledge distillation**, where smaller “student” models mimic larger ones.
* **Caching** and **batching**, processing data more efficiently.
* **Runtime optimizations** with frameworks like ONNX or TensorRT.
* **Speculative decoding**, allowing multiple token predictions in parallel.

In practice, developers balance model accuracy against reduced latency to ensure real-time responsiveness without noticeable loss in quality.

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

From a performance engineering viewpoint, **quantization** works like data compression for AI weights—it converts 32-bit floating-point numbers to lower precisions like 8-bit integers. The result is faster computation, lower memory footprint, and reduced inference cost with minimal quality trade-off.

Meanwhile, **speculative decoding** brings innovation to large language model inference. It pairs a lightweight “draft” model that predicts several tokens ahead with a more powerful “validator” model that confirms or adjusts them. This design lets the system achieve near real-time generation without requiring the heavyweight model to compute every possible output step sequentially.

In practice, many modern chat-based LLMs rely on a combination of both methods for performance tuning.

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

Deploying open models like **Llama**, **Qwen**, or **Mistral** into production environments offers flexibility but also introduces unique challenges.

Common operational hurdles include:

* **High compute overhead**, as these models demand GPU clusters for load handling.
* **Latency management**, especially under unpredictable traffic.
* **Security and compliance concerns**, since you control user data pipelines directly.
* **Maintenance overhead**, requiring constant version monitoring and tuning.
* **Infrastructure complexity**, especially around scaling and monitoring.

From an engineering viewpoint, teams adopting open-source LLMs find that while the freedom to optimize is valuable, it requires disciplined MLOps practices to ensure stability and cost efficiency.

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

**Baseten** and **Modal** let organizations deploy their own inference APIs. Teams can select open-source models, apply hardware-level optimizations, and manage scaling as needed—all while maintaining control over data and costs. In practice, these platforms are often preferred by startups and research teams building domain-specific models.

**Hosted APIs** such as **OpenAI’s** provide managed endpoints where the provider handles scaling, optimization, and availability. This setup is ideal for rapid integration but offers limited flexibility for customization or fine-tuning.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Baseten/Modal** | **OpenAI API** |
| Model ownership | Your own or open-source | Hosted and managed by OpenAI |
| Customization | Highly flexible | Limited |
| Infrastructure | User-managed | Fully managed |
| Data control | Full control | Restricted/third-party managed |
| Setup effort | Moderate | Minimal |

**Section – 2 - Practical Coding Task**

## **1. Optimization Task: Write a Python function that performs a simple matrix multiplication using PyTorch. Then, modify it to use torch.cuda if available to improve speed.**

PyTorch Code

import torch

def matmul\_cpu(A: torch.Tensor, B: torch.Tensor) -> torch.Tensor:

if A.ndim != 2 or B.ndim != 2:

raise ValueError("A and B must be 2D tensors (matrices).")

if A.shape[1] != B.shape[0]:

raise ValueError(f"Incompatible shapes: A{A.shape} and B{B.shape} for matmul.")

return torch.matmul(A, B)

Torch.cuda Code

import torch

def matmul\_auto(A: torch.Tensor, B: torch.Tensor) -> torch.Tensor:

"""

Multiply two 2D tensors (matrices) using CUDA if available, else CPU.

A[m, n] @ B[n, p] -> C[m, p]. Returns result on CPU.

"""

if A.ndim != 2 or B.ndim != 2:

raise ValueError("A and B must be 2D tensors (matrices).")

if A.shape[1] != B.shape[0]:

raise ValueError(f"Incompatible shapes: A{A.shape} and B{B.shape} for matmul.")

device = torch.device("cuda") if torch.cuda.is\_available() else torch.device("cpu")

A\_d = A.to(device, non\_blocking=True)

B\_d = B.to(device, non\_blocking=True)

C\_d = torch.matmul(A\_d, B\_d)

if device.type == "cuda":

torch.cuda.synchronize()

return C\_d.to("cpu", non\_blocking=True)

## **2. Batch Inference Simulation: Simulate running inference on a small text generation model for multiple users concurrently. Optimize your code to handle 100 parallel requests efficiently.**

import asyncio

from concurrent.futures import ThreadPoolExecutor

import torch

from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

MODEL\_NAME = "sshleifer/tiny-gpt2"

tokenizer = AutoTokenizer.from\_pretrained(MODEL\_NAME)

if tokenizer.pad\_token\_id is None:

tokenizer.pad\_token = tokenizer.eos\_token

tokenizer.padding\_side = "left"

model = AutoModelForCausalLM.from\_pretrained(MODEL\_NAME)

device = 0 if torch.cuda.is\_available() else -1

text\_gen = pipeline(

"text-generation",

model=model,

tokenizer=tokenizer,

device=device,

)

def generate\_sync(prompts, max\_new\_tokens=40, batch\_size=8):

return text\_gen(

prompts,

max\_new\_tokens=max\_new\_tokens,

do\_sample=False,

batch\_size=batch\_size,

truncation=True,

pad\_token\_id=tokenizer.pad\_token\_id,

)

async def serve\_100\_requests\_concurrently():

prompts = [f"User {i}: Write one short sentence about concurrency." for i in range(100)]

chunk\_size = 8

chunks = [prompts[i:i+chunk\_size] for i in range(0, len(prompts), chunk\_size)]

loop = asyncio.get\_running\_loop()

with ThreadPoolExecutor(max\_workers=8) as pool:

tasks = [loop.run\_in\_executor(pool, generate\_sync, chunk) for chunk in chunks]

batched\_outputs = await asyncio.gather(\*tasks)

outputs = []

for batch in batched\_outputs:

for item in batch:

if isinstance(item, list):

for seq in item:

if isinstance(seq, dict) and "generated\_text" in seq:

outputs.append(seq["generated\_text"])

elif isinstance(seq, str):

outputs.append(seq)

elif isinstance(item, dict) and "generated\_text" in item:

outputs.append(item["generated\_text"])

elif isinstance(item, str):

outputs.append(item)

return outputs

if \_\_name\_\_ == "\_\_main\_\_":

results = asyncio.run(serve\_100\_requests\_concurrently())

print(f"Generated {len(results)} responses")

for r in results[:5]:

print("-", r[:120], "...")

**3. Quantization Demo: Take a pretrained model from Hugging Face (e.g., distilbert-base-uncased) and apply dynamic quantization. Measure and compare latency before and after quantization.**

import os

import time

import torch

from transformers import AutoTokenizer, DistilBertForSequenceClassification

MODEL\_NAME = "distilbert-base-uncased"

def save\_size\_mb(model) -> float:

"""Rough model size on disk in MB."""

tmp = "tmp\_model\_state.pth"

torch.save(model.state\_dict(), tmp)

size\_mb = os.path.getsize(tmp) / 1e6

os.remove(tmp)

return size\_mb

def measure\_latency\_ms(model, tokenizer, texts, iters=40, warmup=10) -> float:

"""Average per-iteration latency (ms) for forward passes on CPU."""

model.eval()

with torch.no\_grad():

# Warm-up to stabilize caches and JIT paths

for \_ in range(warmup):

inputs = tokenizer(texts, return\_tensors="pt", padding=True, truncation=True)

\_ = model(\*\*inputs)

t0 = time.perf\_counter()

for \_ in range(iters):

inputs = tokenizer(texts, return\_tensors="pt", padding=True, truncation=True)

\_ = model(\*\*inputs)

t1 = time.perf\_counter()

return (t1 - t0) \* 1000 / iters

if \_\_name\_\_ == "\_\_main\_\_":

# Load tokenizer and FP32 model on CPU

tokenizer = AutoTokenizer.from\_pretrained(MODEL\_NAME)

model\_fp32 = DistilBertForSequenceClassification.from\_pretrained(MODEL\_NAME).to("cpu")

# Small input batch to stabilize timing (repeat a few samples)

texts = [

"This was a great experience!",

"I wouldn't recommend this product.",

"Service was quick and friendly."

] \* 8

# Baseline FP32 latency and size

base\_ms = measure\_latency\_ms(model\_fp32, tokenizer, texts)

base\_size = save\_size\_mb(model\_fp32)

print(f"FP32 avg latency: {base\_ms:.2f} ms | size: {base\_size:.1f} MB")

# Apply dynamic quantization to Linear layers (INT8 weights, dynamic activations)

q\_model = torch.quantization.quantize\_dynamic(

model\_fp32,

{torch.nn.Linear}, # target Linear layers (typical for transformers)

dtype=torch.qint8

)

# Quantized latency and size

q\_ms = measure\_latency\_ms(q\_model, tokenizer, texts)

q\_size = save\_size\_mb(q\_model)

print(f"INT8 avg latency: {q\_ms:.2f} ms | size: {q\_size:.1f} MB")

# Report speedup and compression

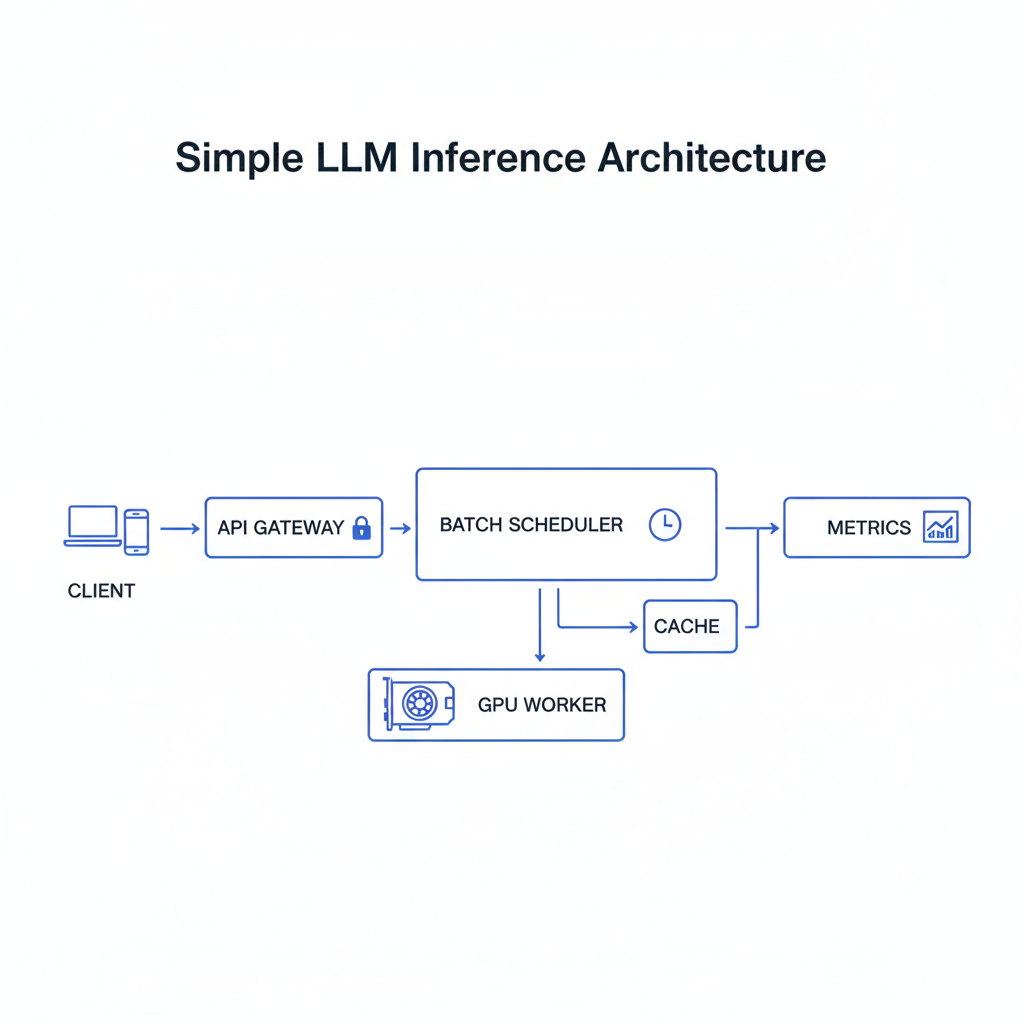
if q\_ms > 0:

print(f"Speedup: {base\_ms/q\_ms:.2f}x | Size reduction: {base\_size/q\_size:.2f}x")

**Section – 3 - System Design Challenge**

## **1. Design a scalable inference system that can serve millions of LLM requests per day with <100ms latency. Your design should include:** **- Model deployment architecture** **- Load balancing approach** **- Caching and batching strategies** **- Monitoring and fallback plan**

## **Model deployment architecture**



* Inference stack
  + Tokenizer/gateway: Stateless API layer (gRPC/HTTP/2) for auth, request shaping, token limits, rate limiting, and routing by model/version/region. Keep small and horizontally scalable.
  + Scheduler/serving runtime: An inference server that supports continuous batching, KV-cache reuse, paged attention, and tensor parallel where needed. Examples include high-throughput runtimes that implement efficient request scheduling and memory paging.
  + GPU workers:
    - Single-GPU replicas for small/medium LLMs to simplify scheduling and reduce cross-GPU latencies.
    - Multi-GPU tensor/PP sharding only for very large models that cannot fit on one device; prefer tensor parallel ≤2 to limit communication overhead.
  + Storage and artifacts:
    - Model registry (immutable versions, signed artifacts).
    - Feature flags/config store for runtime toggles (batch size caps, draft model on/off).
    - Warm pools with preloaded weights to avoid cold-start penalties.
* Performance optimizations
  + Quantization: INT8/FP8 or weight-only quantization for throughput and VRAM savings; validate accuracy regressions with A/B.
  + Mixed precision: FP16/BF16 with autocast; enable flash attention and fused kernels.
  + Speculative decoding: Pair a small “draft” model with the main model to reduce per-token compute, especially for short-medium outputs.
  + KV cache + paged attention: Reuse prefill computations and minimize memory movement, keeping TTFT low.

## **Load balancing approach**

* Global routing
  + Anycast or DNS latency-based routing across regions; route users to nearest healthy region to minimize network RTT.
  + Weighted shifting for canaries and blue/green rollouts; drain connections gracefully during deploys.
* Regional balancing
  + Layer 4/7 load balancer with:
    - Sticky routing by model/version to keep cache locality (KV/page cache and tokenizer cache).
    - Queue-aware routing to avoid overloading a hot replica; route based on queue depth, in-flight tokens/sec, and available KV memory.
* Backpressure and shedding
  + Token budget enforcement per request to fit the <100 ms SLO.
  + Early reject or enqueue-with-timeout; return 429 with retry-after to keep tail latency controlled.
  + Priority queues for internal/paid tiers; throttle low-priority traffic during spikes.

## **Caching and batching strategies**

* Caching
  + Prompt normalization and hashing: Lowercase, strip whitespace, canonicalize system prompts; hash to identify repeated requests.
  + Output cache:
    - Exact match cache for deterministic decoding (greedy/top-1).
    - Segment cache for system prompts and few-shot contexts; reuse prefill KV states for repeated prefixes.
  + Tokenizer cache: Persist piecewise tokenization for popular prompts/templates to avoid repeated CPU-bound work.
* Batching
  + Continuous batching at the server: Aggregate requests arriving within a short window (e.g., 1–5 ms) into micro-batches to maximize GPU utilization without visible latency impact.
  + Left padding for causal models: Ensures generation focuses on the rightmost active tokens, reducing wasted computation during batch decoding.
  + Dynamic batch sizing: Adjust by live metrics (TTFT, decode tokens/sec, VRAM headroom). Cap batch size to prevent GPU OOM and long-tail spikes.
* Response shaping
  + Small max\_new\_tokens for real-time endpoints (e.g., 16–64) to keep under 100 ms; provide a separate “bulk” endpoint for longer generations.
  + Streamed responses for interactive use, but ensure first-chunk time remains <100 ms by optimizing prefill.

## **Monitoring and fallback plan**

* SLOs and key metrics
  + Latency: p50/p90/p95 end-to-end; TTFT and TBT (time-between-tokens) separately.
  + Throughput: tokens/sec per replica and region; queue depth and admission rate.
  + GPU/host health: SM utilization, memory used/free, memory fragmentation, kernel time, PCIe/NVLink bandwidth.
  + Errors: 4xx/5xx rates, OOM, timeouts, token budget violations.
  + Cache metrics: hit/miss for prompt and output caches; KV page hit rate; tokenizer cache hits.
* Observability
  + Tracing: Correlate gateway → scheduler → GPU kernel spans; include prompt length, tokens generated, and batch size to explain tail latency.
  + Log sampling with PII redaction; keep per-request token counts for cost attribution and anomaly detection.
* Autoscaling
  + Scale on composite signals: in-flight tokens, queued requests, TTFT drift, GPU memory headroom.
  + Warm standby replicas with pre-loaded weights to avoid cold-start penalties; target <5 s to “hot ready.”
* Fallbacks and resilience
  + Tiered degradation:
    - Step 1: Reduce max\_new\_tokens; Step 2: tighten batch caps to reduce queue time; Step 3: enable more aggressive quantization; Step 4: switch to a smaller model class for overload or regional failure.
  + Multi-region failover with state-light requests; keep model versions synchronized across regions.
  + Circuit breakers at gateway per model/version/region; automatic ejection of flapping replicas.
  + Optional third-party hosted API fallback for critical paths with strict budget guardrails and clear data controls.

## **Reference sizing and SLO guidance (example)**

* Traffic: 3–5 million requests/day with short prompts (≤256 tokens) and small completions (≤32 tokens).
* Hardware: A100/H100 or comparable; aim for single-GPU replicas per model variant to reduce interconnect overhead.
* Targets:
  + TTFT p95 ≤ 50 ms, end-to-end p95 ≤ 100 ms for short responses.
  + Prefill throughput tuned via micro-batch windows (1–5 ms); decode throughput sustained via speculative decoding and cache reuse.
* Practices:
  + Keep separate “realtime” and “bulk” endpoints with different token budgets, schedulers, and batch caps.
  + Pre-warm popular model versions and system prompts; rotate keys and configs via feature flags without restarts.

## **Deployment blueprint (paste-friendly outline)**

* Gateway
  + AuthN/Z, quotas, token budgets, request normalization
  + Latency-aware routing to regional front doors
* Regional front door
  + L7 LB with sticky routing by model/version
  + Queue-aware routing using replica health and queue depth
* Inference tier
  + Scheduler with continuous batching and KV/paged attention
  + GPU workers (single-GPU preferred), mixed precision + quantization
  + Speculative decoding toggle (feature flag)
* Caching tier
  + Prompt/output cache (Redis/Redis-like), tokenizer cache
  + KV prefix reuse for frequent system prompts/templates
* Control plane
  + Model registry, version pinning, warm pools, feature flags
  + Autoscaler driven by tokens/sec, TTFT, queue depth, VRAM headroom
* Observability
  + Metrics: latency (TTFT, p95), tokens/sec, GPU utilization, cache hit rates
  + Tracing across gateway → scheduler → kernel
  + Alerting on SLO breaches; circuit breakers and auto-eject
* Resilience
  + Multi-region active/active, failover runbooks
  + Progressive degradation steps and smaller-model fallback
  + Optional external API fallback with strict budgets and redaction

**Section – 4 – Research and Innovation**

**1. Recent paper on speculative decoding. Summarize your understanding in 200–300 words**

Summary -

Speculative decoding accelerates autoregressive LLM inference by pairing a fast “draft” generator with a larger “target” model that verifies the drafted tokens, yielding identical output distributions to standard decoding while reducing wall-clock time per token. Recent work extends this paradigm along two directions: better draft-target coordination and removing the separate draft altogether. Single-model approaches (e.g., Medusa, EAGLE variants, Hydra) add auxiliary heads or layers to the target so it can propose multi-token continuations in parallel, increasing acceptance without managing a second model process; these designs trade additional target compute for simpler deployment while preserving distributional correctness guarantees under the verification step. System-oriented studies show practical speedups of roughly 2–3× for short and medium continuations when the draft is well-aligned and hardware-efficient, with further gains from overlapping draft generation and target verification, paged attention, and careful scheduling that converts draft capacity into higher acceptance rates rather than added latency. Industry reports highlight vocabulary-agnostic schemes that decouple the draft from target tokenization, broadening compatibility and easing integration across heterogeneous model families while sustaining 2–3× improvements on production tasks. At scale (e.g., Llama serving), teams combine speculative decoding with quantization, mixed precision, and continuous batching to keep time-to-first-token and p95 latencies low under multi-tenant load, emphasizing A/B validation to monitor quality drift and acceptance dynamics across prompts and temperatures. Open challenges include maintaining high acceptance when prompts are out-of-distribution, balancing draft cost vs. acceptance under different sampling settings, and integrating speculation with KV-cache reuse and streaming without inflating tail latency.

**Section – 5 – Build Something**

**Github Link - https://github.com/Divyanshmishra101010/resume-analyzer**