## **Section 3 – System Design Challenge**

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

**Solution:**

# **1 — Model deployment architecture**

## **High-level components**

* **API Gateway / Edge** (Envoy/NGINX/Kong): TLS, auth, rate limits, routing, global LB.
* **Ingress / Front Proxy**: performs request classification (priority, user type), small validation, and forwards to internal gateway.
* **Request Orchestration Layer**:
  + **Router / Scheduler** (stateless): chooses which model pool to route a request to (model version, region, priority).
  + **Queue / Broker** (Redis Streams / Kafka / NATS): short-lived queue for backpressure, priority queues for latency-sensitive vs batch jobs.
* **Model Serving Layer**:
  + **Model Replica Pools** (containers / pods): each pool runs one model version.
    - For small/medium models (≤ 13B) you can run 1 replica per GPU.
    - For large models (> 30B) use model parallelism (tensor/pipeline parallel) across multiple GPUs per replica.
  + **Triton / TorchServe / Ray Serve / HF Inference** as serving runtime.
  + **Inference optimizations**: TensorRT / ONNX + INT8/FP16 quantization (or bitsandbytes 4-bit), DeepSpeed-Inference, custom CUDA kernels when needed.
* **Embedding & Cache Services**:
  + **Embedding service**: small model to compute vectors (MiniLM/Muennich).
  + **Vector DB** (FAISS / Milvus / RedisVector) for semantic cache lookup.
  + **Exact-response cache**: Redis (hot), backed by persistent DB (sqlite or key-value store).
* **Auxiliary**: metrics ingestion (Prometheus), tracing (Jaeger), logging (ELK), alerting.

## **Model placement & parallelism**

* **Small models (<=13B)**: single-GPU replicas; horizontal scale.
* **Medium (13B–30B)**: consider 2–4 GPU TPU slices, or quantize + run single-GPU.
* **Large (>30B)**: use tensor + pipeline parallelism with DeepSpeed/torch.distributed; each logical replica = multiple GPUs.
* **Hybrid:** keep a smaller, quantized copy (7B/2–4bit) as a “fast-fallback” pool.

# **2 — Load balancing approach**

## **Multi-layer LB strategy**

1. **Global DNS LB** (optional): route user to nearest region for lower RTT (Cloudflare / Route53 Geolocation).
2. **Edge Gateway (L7)**: terminates TLS, does auth & rate-limiting, rejects malformed requests early.
3. **Internal Router**:
   1. Maintains per-pool **health** and **load metrics** (GPU utilization, queue length, current batch sizes).
   2. Routes by: model version → region → priority → least-loaded replica.
   3. Use **consistent hashing** for session affinity only if you keep ephemeral session state (prefer stateless).
4. **Per-Pod LB / Intra-pool**:
   1. Use **round-robin + least-queue-length** strategy. Prefer sending to replicas with smallest request queue.
   2. Health checks (liveness / readiness) and circuit breaker to avoid sending to slow nodes.
5. **Backpressure & Priority**:
   1. If system overloaded, queue lower-priority jobs or route to batch-only pool.
   2. Use token-bucket rate limiting per user/API-key.

## **Practical tech**

* Envoy or NGINX at the front, internal gRPC/HTTP with service mesh (Istio, Linkerd) for observability/traffic control.
* Router can be simple service reading Redis stats or using metrics API from Triton.

# **3 — Caching and batching strategies**

## **Caching (multi-tier)**

1. **Exact-match LRU cache (in-memory / Redis)**
   1. Key = hashed(normalized\_prompt + params). Return full response.
   2. TTL depends on freshness; for deterministic responses can be long.
2. **Request coalescing (de-duplication)**
   1. If identical requests arrive concurrently, coalesce them so only one inference runs; others wait for result.
3. **Semantic cache (vector similarity)**
   1. Embed incoming prompt (short context) and lookup top-k in FAISS/RedisVector. If similarity > threshold, return cached answer or initiate re-ranking/fallback.
4. **KV-cache reuse (LLM internals)**
   1. For streaming tokens, keep attention KV caches per session so incremental token generations avoid recomputing earlier layers.
5. **Partial result cache**
   1. Cache embeddings or intermediate responses for multi-turn dialogues.

## **Batching approaches**

* **Dynamic batching** at model server (recommended: NVIDIA Triton or custom batcher)
  + Parameters:
    - max\_batch\_size (e.g., 4–32 depending on model and GPU)
    - max\_batch\_wait\_ms (e.g., 5–20 ms)
    - priority-aware batching (latency-sensitive requests bypass larger batch waits)
  + Use adaptive algorithms: increase batch window when QPS is high; shrink when latency approaches SLO.
* **Length-based bucketing**: group inputs by token-length to minimize padding overhead.
* **Hybrid (async coalescing + sync batch run)**:
  + Queue incoming requests into an in-memory ring buffer.
  + Every T ms or when BATCH\_SIZE reached, create a real batch and call model once.
* **Speculative Decoding**:
  + Run small draft model to generate k tokens quickly; verify with main model in a single forward pass (drastically reduces main-model forward passes per token).
* **Streaming tokens**:
  + Use token streaming for large outputs to reduce perceived latency; but ensure end-to-end P95 stays <100ms where required.

### **Tuning knobs for <100ms P95**

* Keep max\_batch\_wait\_ms very low (e.g., 5–12 ms) for interactive scenarios.
* Use many small replicas rather than few large ones to reduce queueing.
* Use quantization & optimized kernels so single forward pass is <~15–30ms for short contexts.

# **4 — Monitoring & fallback plan**

## **Observability**

* **Metrics (Prometheus)**:
  + Request rate (RPS), success/error count
  + Latencies: P50/P90/P95/P99 (end-to-end + model inference only)
  + GPU metrics: utilization %, memory usage, SM occupancy, batch sizes
  + Queue lengths, batch\_wait\_ms, cache hit/miss rates
* **Tracing (Jaeger/OpenTelemetry)**:
  + Trace end-to-end request path: gateway → router → worker → cache → db
  + Span timing for queue wait, batch time, forward pass time, post-processing
* **Logging**:
  + Structured logs with request-id, model-version, user-id, latency
* **Dashboards & Alerts (Grafana)**:
  + SLO dashboard: P95 <100ms
  + Alerts:
    - P95 > 100ms for > 2 minutes
    - GPU memory OOM / high error rate
    - Cache miss rate suddenly increases

## **Health checks & automated responses**

* Liveness/readiness probes per pod.
* Automated scale-out when GPU utilization > X% for Y seconds or request queue length > Z.
* Circuit-breakers: if a worker’s P95 exceeds threshold, mark unhealthy and drain.

## **Fallbacks & graceful degradation**

1. **Primary → Fallback model**:
   1. If main model overloaded or failing, route to smaller quantized model (7B / distilled) — serves with lower quality but fast.
2. **Cache fallback**:
   1. If a request matches exact/semantic cache, serve immediately.
3. **Reduced feature-mode**:
   1. Disable expensive features (long-context, tool-calls) when overloaded.
4. **Rate-limit & backpressure**:
   1. Throttle non-critical users; return HTTP 429 for excess.
5. **Queued responses**:
   1. For non-interactive requests, allow longer queueing and later callback/webhook.
6. **Graceful error response**:
   1. Provide short canned reply + retry token for clients when full failure occurs.