Backend Developer Assignment Submission

Name : Aravind Reddy Chittemreddy

Date : 8th November 2025

Role : Junior Backend Developer

Introduction

This document outlines my design and implementation approach for building a scalable, efficient, and cost-effective backend for a machine translation system. It covers high-level architecture, scalability strategies, performance targets, API design, database schema, priority handling, monitoring, and code implementation details.

High Level Design

1. System Architecture
    - Use GPU-enabled servers to handle heavy neural translation model inferences, as GPUs provide significant speed advantages over CPUs.
    - Deploy the backend in containerized environments managed by Kubernetes, enabling flexible scaling and ease of maintenance.
    - Implement a load balancer to distribute incoming translation requests evenly across multiple server instances, ensuring fault tolerance and balanced load.
    - Store user information and translation metadata in a NoSQL database such as MongoDB, for fast read/write operations and flexible data schemas.
    - Utilize object storage services like Amazon S3 to securely store large documents and translation archives.

2. Scalability
    - The system will scale horizontally by adding more GPU container instances dynamically based on incoming translation request volume using auto-scaling.
    - During idle or low-demand periods, the backend will scale down by shutting down unnecessary instances to reduce costs, while maintaining a small pool of warm instances to reduce startup latency.
    - An in-memory cache like Redis will store frequent translation results to serve repeated requests instantly and reduce backend load.
    - By combining horizontal scaling, dynamic resource management, and caching, the system will handle increased user load efficiently and cost-effectively.

3. Performance Requirements
    - To meet the target translation speed of 1,500 words per minute, the system will batch process smaller chunks of text, enabling parallel inference on multiple GPU instances.

- For translating large documents (e.g., 15,000 words within 12 minutes), the workload will be split and distributed across several GPUs to be translated concurrently, ensuring timely completion during peak load.
- GPU and CPU allocation will be managed dynamically based on job size and priority to optimize resource utilization without wasting compute power.
- Model optimization techniques such as quantization will be applied to reduce GPU memory consumption, enabling efficient handling of large model sizes (60–100GB).
- Resource pooling and job queuing strategies will be used to minimize idle GPU/CPU time and reduce operational costs during low-demand periods.

4. Priority Handling
    - I will implement a priority queue where each translation request is assigned a priority level such as high, medium, or low.
    - High-priority requests will be placed in a separate queue or tagged to ensure they are processed ahead of lower-priority jobs.
    - The system will support preemption so urgent jobs can jump the queue and be handled immediately.
    - Compute resources like GPUs will be allocated dynamically with preference given to higher priority tasks to minimize their wait time.
    - I will also add fairness mechanisms to prevent starvation of lower-priority requests, for example by limiting maximum waiting times.

5. Monitoring and Cost Estimates
    - I will use monitoring tools like Prometheus and Grafana to track system health and performance metrics such as GPU/CPU utilization, API request latency, and translation queue lengths.
    - Alerts will be set up to notify the team if any metric deviates from normal thresholds, for example, if GPU usage is unusually low during peak hours or if API latency spikes.
    - For cost estimation, I will calculate based on expected translation volumes: handling 10,000, 100,000, and 1 million words per day, factoring in GPU compute hours, storage needs, and network costs.
    - Using cloud pricing models, I will estimate costs by considering TFLOPS per GPU hour, average model runtime per word, and storage costs for user data and translated documents.
    - Sample log entry might look like: "[Timestamp] GPU Utilization: 85%, API Latency: 150ms, Queue Length: 20, Job Priority: High" to provide detailed insight.
    - This combined monitoring approach helps maintain system reliability and cost-effectiveness by enabling proactive resource management and budget control.

Low Level Design

1. API Design
    - Design RESTful APIs with the following endpoints:

    - POST /translate : Accepts new translation requests containing source text, source language, target language, and an optional priority flag.
    - GET /translate/{jobID}/status : Returns the current status of the translation job, such as pending, in-progress, or complete.
    - GET /translate/{jobID}/result : Fetches the translated document once the job is finished.
- Implement JWT-based authentication to ensure secure access to the APIs.
- Use rate limiting (e.g., 100 requests per minute per user) to prevent abuse and ensure fair usage.
- Validate all incoming requests for correctness of data formats and required fields before processing.
- Log API request and response metrics to monitor performance and identify issues.

2. Database Schema
    - Users table : Stores user information with fields such as userID (primary key), email, hashedPassword and createdAt.
    - TranslationJobs table : Tracks translation requests with fields including jobID (primary key), userID (foreign key), sourceLanguage, targetLanguage, jobStatus (e.g., pending, in-progress, completed), priorityLevel, createdAt and completedAt.
    - TranslationMetadata table: Contains metadata for each job such as jobId (foreign key), wordCount, translationDuration, and a cacheHit flag indicating if the result was served from cache.
    - PriorityQueue table: Maintains job prioritization with fields like jobId, priority, and enqueueTimestamp to facilitate ordered job processing.

This schema efficiently supports user tracking, job management, metadata storage, and priority handling for the translation backend.

3. Code Implementation (Pseudocode)
    - I will use a task queue system where translation requests are submitted with a priority tag and enqueued accordingly.
    - A worker process will continuously poll the priority queue to fetch the highest priority job available.
    - The worker will allocate necessary GPU/CPU resources, perform the translation, save the result, and release resources back to the pool.
    - Autoscaling logic will monitor current system load and active instances; it will launch new instances if utilization is above a threshold and shut down idle instances to optimize costs.
    - Example Pseudocode:

```
class TranslationSystem:
    def submit_job(self, request):
        job = self.create_job(request)
        self.priority_queue.put((job.priority, job))
        return job.id
    def process_jobs(self):
        while True:
            priority, job = self.priority_queue.get()
            self.allocate_resources(job)
            translated_text = self.translate(job.text,
job.source_lang, job.target_lang)
            self.save_translation(job.id, translated_text)
            self.release_resources(job)

def auto_scale(current_load, active_instances):
    if current_load > 80:
        launch_instance()
    elif current_load < 20 and active_instances > 1:
        shutdown_instance()
```

Conclusion

In summary, the proposed backend design balances performance, scalability, and cost by leveraging GPU resources, container orchestration, caching, and priority queues. With careful monitoring and dynamic scaling, it can reliably meet translation demands while optimizing resource usage. This approach reflects a practical, maintainable solution to support a growing user base and workload.