Rapport d'Architecture

LLM — Architecture parallèle (Multi-serveurs / Multi-clients)

Auteurs : Yazid Dhouioui & Mazen Haouari

Date : 6 octobre 2025

Table des matières

1 Introduction

Ce document décrit une architecture client–serveur parallèle optimisée pour un service basé sur des modèles de langage (LLM). Le design cible la **scalabilité horizontale**, la **faible latence**, la **haute disponibilité** et la **sécurité**. Les composants essentiels sont : clients multiples, API Gateway, Load Balancer, pool d'inférence (plusieurs serveurs GPU/CPU), services d'embeddings et de retrieval, base vectorielle, cache, orchestrateur et observabilité.

2 Vue d'ensemble

Principes architecturaux :

- **Séparation des responsabilités** : gateway (auth, TLS, quotas), load balancer (distribution), inférence (LLM), retrieval (RAG) et stockage.
- **Scalabilité horizontale** : multiples instances d'inférence, sharding de la vector DB, réplicas.
- **Optimisations** : batching dynamique, quantization, cache des réponses fréquentes.
- **Résilience** : redondance, health checks, autoscaling.

3 Diagramme d'architecture

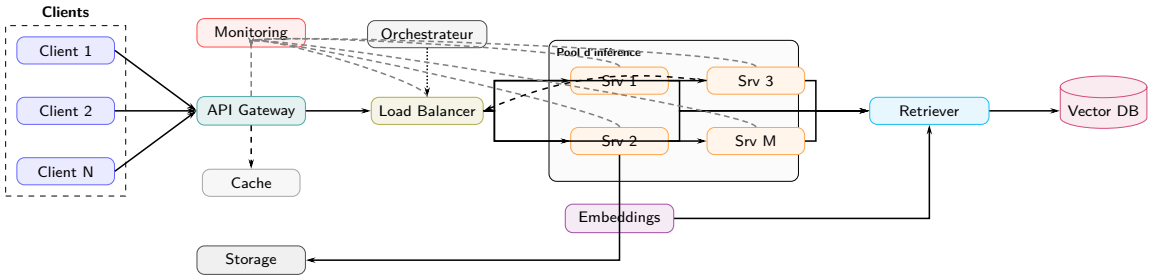


FIGURE 1 – Architecture parallèle optimisée (multi-clients, multi-serveurs, Gateway + Load Balancer) pour un LLM.

4 Description et justification de l'architecture

L'architecture présentée est une **architecture parallèle client-serveur distribuée** optimisée pour le déploiement et l'exploitation d'un modèle de langage de grande taille (LLM). Elle combine plusieurs couches fonctionnelles — clients, passerelle (gateway), équilibrage de charge, serveurs d'inférence parallèles, et backend vectoriel — afin d'assurer **performance, tolérance aux pannes et scalabilité horizontale**.

4.1 Composants principaux

- **Clients** : représentent les utilisateurs finaux ou les applications externes qui envoient des requêtes au modèle (chatbot, API d'analyse, interface web, etc.). Plusieurs clients peuvent être connectés simultanément grâce à la structure parallèle et à la gestion du trafic via la passerelle et l'équilibrage de charge.

— **API Gateway** : c'est le point d'entrée unique du système. Elle centralise la réception des requêtes, gère l'authentification, la journalisation et le routage des appels vers le load balancer. Elle permet également d'appliquer des politiques de sécurité et de contrôler le flux réseau.
— **Load Balancer** : répartit intelligemment les requêtes entre les différents serveurs d'inférence pour optimiser la charge de calcul. Cela évite la saturation d'un serveur et améliore la latence moyenne. Le load balancer peut fonctionner selon différentes stratégies (round-robin, least connections, etc.).
— **Serveurs d'inférence (Srv 1 à Srv M)** : ce sont les nœuds principaux du calcul parallèle. Chaque serveur exécute une instance du LLM (ou une portion du modèle shardée sur GPU). En parallèle, ils assurent la génération, l'embedding ou l'évaluation des réponses. Le dimensionnement de ce pool permet de répondre à des milliers de requêtes concurrentes.
— **Module d'Embeddings** : convertit les textes en vecteurs sémantiques, utilisés pour les recherches contextuelles ou les mécanismes de retrieval-augmented generation (RAG).
— **Retriever** : effectue la recherche de contexte pertinent dans le *Vector DB* à partir des embeddings. Il enrichit la requête utilisateur avant passage au modèle d'inférence.
— **Vector DB** : base de données vectorielle (comme `FAISS`, `Milvus` ou `Pinecone`) stockant les représentations vectorielles des connaissances. Elle permet des recherches rapides par similarité cosinus ou euclidienne.
— **Cache** : permet de réduire les temps de réponse en stockant temporairement les résultats récents ou fréquemment demandés (ex. Redis). Il soulage les serveurs d'inférence et diminue la latence moyenne.
— **Orchestrateur** : gère la supervision et le déploiement des serveurs d'inférence. Il peut être basé sur `Kubernetes`, `Docker Swarm` ou tout autre outil de conteneurisation pour assurer l'auto-scaling, la mise à jour continue et la reprise automatique en cas de panne.
— **Monitoring** : assure le suivi de la santé des nœuds, du taux d'erreur, de la latence et de l'utilisation des ressources. Permet une maintenance proactive.
— **Storage** : stockage persistant pour les logs, les modèles, les checkpoints et les jeux de données utilisés pour l'entraînement ou la personnalisation.

4.2 Pourquoi cette architecture est-elle optimale ?

— **Scalabilité horizontale** : de nouveaux serveurs d'inférence peuvent être ajoutés dynamiquement sans interrompre le service, ce qui permet de gérer la montée en charge.
— **Haute disponibilité** : grâce au load balancer et à l'orchestrateur, la panne d'un serveur n'interrompt pas le fonctionnement global. Les requêtes sont automatiquement redirigées vers les nœuds actifs.
— **Parallélisme massif** : plusieurs serveurs peuvent exécuter des inférences simultanément, réduisant drastiquement la latence moyenne par requête.
— **Optimisation du trafic** : la présence du cache et du gateway réduit le nombre de traitements inutiles côté serveur, tout en maintenant la cohérence et la sécurité.
— **Adaptabilité aux LLM modernes** : cette architecture est compatible avec les modèles distribués (comme DeepSeek, GPT, ou Llama 3) et supporte aussi bien

le partitionnement par couches (tensor parallelism) que le sharding par données (data parallelism).

— **Observabilité et résilience** : la couche de monitoring permet de détecter rapidement les anomalies, tandis que l'orchestrateur garantit le redémarrage automatique des conteneurs défaillants.

En somme, cette architecture parallèle est dite **optimale** car elle concilie *efficacité de traitement*, *sécurité*, *évolutivité* et *robustesse*, tout en restant suffisamment modulaire pour accueillir différents types de modèles et d'applications.

5 Mise à jour vers un système réparti

La mise à jour de cette architecture vers un système réparti présente un intérêt majeur pour garantir la scalabilité, la résilience et la performance du modèle de langage à grande échelle (LLM). Cette évolution apporte plusieurs avantages clés :

— **Scalabilité accrue :** la répartition des charges de calcul sur plusieurs nœuds physiques ou virtuels permet de traiter simultanément un volume beaucoup plus important de requêtes clients, tout en minimisant les temps de latence.

— **Résilience améliorée :** en cas de défaillance d'un serveur, les autres nœuds du système peuvent automatiquement prendre le relais, garantissant ainsi la continuité du service sans interruption majeure.

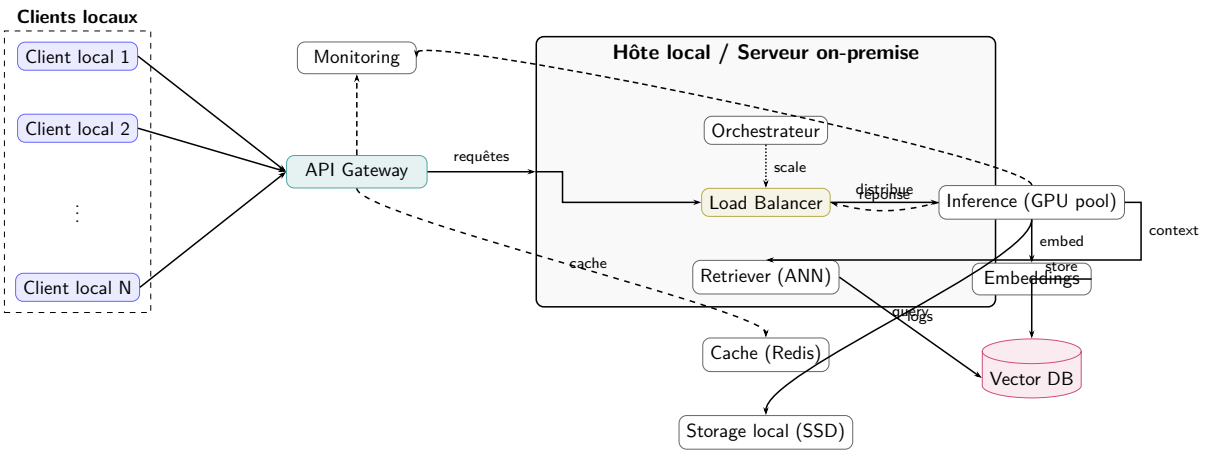
— **Performance optimisée :** la parallélisation des traitements permet une exécution plus rapide des requêtes et un meilleur temps de réponse global du système.

— **Maintenance facilitée :** chaque composant du système peut être mis à jour, remplacé ou amélioré indépendamment, sans impacter l'ensemble de l'architecture.

Ainsi, l'adoption d'un système réparti constitue une étape stratégique pour soutenir l'évolution et la montée en charge des modèles de langage modernes.

6 Architecture locale (déploiement on-premises / single-host)

Diagramme de l'architecture locale optimisée



Description synthétique

Cette architecture locale regroupe l'ensemble des composants sur un hôte (ou un petit cluster local) : point d'entrée (API Gateway), équilibrage local des requêtes (Load Balancer), pool d'inférence utilisant un ou plusieurs GPU, service d'embeddings, retriever connecté à une base vectorielle locale, cache Redis, stockage persistant sur SSD et outils d'observabilité. L'orchestrateur est léger (par ex. `docker-compose` ou des unités `systemd`) pour gérer les conteneurs et les redémarrages locaux.

Pourquoi choisir une architecture locale ?

- **Contrôle des données :** les données restent sur site (utile pour la confidentialité ou contraintes réglementaires).
- **Coûts prévisibles :** absence de facturation cloud continue ; coût matériel unique.
- **Latence faible pour les clients locaux :** trafic sur le réseau local réduit la latence réseau.
- **Simplicité opérationnelle** pour des environnements de développement, tests ou déploiements restreints.

Contraintes et recommandations

- **Scalabilité limitée :** montée en charge verticale (ajout de GPU/CPU/SSD) mais scalabilité horizontale plus coûteuse et complexe.
- **Résilience matérielle :** faut prévoir redondance (RAID, nœud de secours) ou procédures de reprise pour tolérer les pannes physiques.
- **Maintenance et mises à jour** : automatiser via scripts/CI pour déployer des images Docker et appliquer les patches.
- **Optimisation GPU** : utiliser batching adaptatif, quantization et moteurs optimisés (TensorRT / ONNX) pour maximiser le throughput sur ressources locales.
- **Sauvegardes régulières** : configurer sauvegardes du Vector DB et des checkpoints vers un stockage hors site si nécessaire.

Notes de déploiement pratique

- Déploiement léger : `docker-compose.yml` pour orchestrer Gateway, Redis, Vector DB (Milvus/FAISS dans conteneur), inférence (containerized runtime), Prometheus et Grafana.
- Pour production locale : considérer un petit cluster (2+ hôtes) avec un orchestrateur (Kubernetes) si montée en charge ou haute disponibilité requises.
- Surveillance : exporter métriques GPU (nvidia-smi exporter), logs centralisés et alerting pour éviter les interruptions.

6.1 Architecture locale intelligente