# FIRE-LLM : Réordonnancement efficace de contenus pour de la recherche d'information avec LLM

Louis Remy[1,3], Sandra Bringay[1,2], Pascal Poncelet[1], Maximilien Servajean[1,2]

[1] LIRMM - Université de Montpellier, CNRS, France
[2] AMIS - Université de Montpellier Paul-Valéry, France
[3] ACELYS - Montpellier, France

{prénom.nom}@lirmm.fr

## Résumé

*Les grands modèles de language (LLM) sont utilisés pour améliorer le classement de contenus en recherche d'information, en demandant au modèle de produire directement une liste ordonnée de candidats. Lorsque la liste est longue, cette stratégie devient coûteuse et plus instable (e.g. classements tronqués), avec une latence qui augmente fortement. Cet article présente Filter and Rerank Efficiently with LLM (FIRE-LLM), une chaîne en trois étapes : (i) récupération initiale par BM25, (ii) filtrage par LLM afin de réduire l'ensemble de candidats, puis (iii) production du classement final en une seule passe sur la liste filtrée. Cette organisation vise à limiter le nombre d'inférences et à éviter le traitement direct de listes très longues. L'évaluation zero-shot est menée sur TREC Deep Learning 2019 et TREC Deep Learning 2020. Notre approche est comparée à l'état de l'art et les résultats rapportés selon la métrique NDCG@10 indiquent des performances comparables, tout en réduisant le coût d'inférence. Les mesures de temps ainsi que les estimations d'émissions de $CO_2$ montrent un gain croissant avec la taille de la liste candidate.*

## Mots-clés

*Recherche d'information, analyse énergétique, grands modèles de langage.*

## Abstract

*Large language models (LLMs) are used to improve the ranking of content in information retrieval by asking the model to directly produce an ordered list of candidates. When the list is long, this strategy becomes costly and less stable (e.g. truncated rankings), with latency greatly increasing. This paper presents Filter and Rerank Efficiently with LLM (FIRE-LLM), a three-stage pipeline : (i) initial retrieval with BM25, (ii) LLM-based filtering to reduce the candidate set, and (iii) generation of the final ranking in a single pass over the filtered list. This design aims to limit the number of LLM inferences and to avoid processing very long lists directly. Zero-shot evaluation is conducted on TREC Deep Learning 2019 and TREC Deep Learning 2020. We compare our approach to the state of the art, and results reported with NDCG@10 show comparable effectiveness while reducing inference cost. Runtime measurements and $CO_2$ emission estimates further indicate that the gains increase with the size of the candidate list.*

## Keywords

*Information retrieval, energy consumption analysis, large language models.*

## 1 Introduction

Les grands modèles de langage (LLM) sont de plus en plus utilisés en recherche d'information (RI) à différentes étapes du pipeline, notamment pour la réécriture de requête, la recherche, le réordonnancement et la formulation de réponse. Parmi ces usages, nous nous intéressons plus particulièrement au réordonnancement, qui consiste à réviser l'ordre des passages candidats afin de mieux refléter leur pertinence pour la requête. Dans ce cadre, nous étudions plus précisément l'emploi des LLM en réordonnancement *zero-shot* [1, 9, 16]. Dans certaines configurations, notamment lorsque les modèles traitent plusieurs passages dans un même contexte, la qualité du réordonnancement est améliorée par rapport aux méthodes classiques basées sur la similarité vectorielle.

Les approches de réordonnancement à l'aide de LLM se déclinent généralement en trois paradigmes :

— *Pointwise reranking* (réordonnancement point par point) : une fonction de score $f$ prend en entrée la requête $q$ et un seul passage $p_i$, puis lui associe un score $s_i = f(q, p_i)$. Les passages sont ensuite triés selon ce score ;

— *Pairwise reranking* (réordonnancement par paire) : une fonction d'évaluation compare deux passages $p_i$ et $p_j$ relativement à la requête $q$, et détermine lequel doit être préféré. Formellement, elle peut être vue comme une fonction $f(q, p_i, p_j)$ produisant une préférence binaire ou un score de préférence. Un algorithme de tri fondé sur ces comparaisons permet ensuite d'ordonner la liste ;

— *Listwise reranking* (réordonnancement par liste) : le modèle agit comme une fonction globale $f$ qui prend en entrée la requête $q$ et l'ensemble complet des $n$ passages candidats $S = \{p_1, \ldots, p_n\}$,

afin de produire directement une séquence ordonnée $\pi$, c'est-à-dire une permutation des passages : $f(q, S) = (p_{\pi(1)}, \ldots, p_{\pi(n)})$. Cette approche permet de capturer les interactions entre passages et peut, dans certains cas, réduire le coût global du réordonnancement en évaluant plusieurs passages au cours d'une même inférence.

Dans ce travail, nous choisissons le paradigme du *listwise reranking* en raison de son efficacité [6, 12, 17]. Nous mettons néanmoins en évidence ses limites lorsque la liste candidate devient trop grande, typiquement au-delà d'une trentaine de passages. Bien que les LLM modernes disposent de fenêtres de contexte suffisamment larges (par exemple 32 768 tokens) pour prendre en entrée un grand nombre de passages, la difficulté principale ne vient pas de cette capacité. Elle vient plutôt du fait que le modèle raisonne moins bien lorsque trop de passages sont présentés en une seule fois. En pratique, lorsque le nombre de passages augmente, le modèle respecte moins souvent le format de sortie attendu et produit plus fréquemment des listes tronquées, des identifiants invalides ou répétés, voire des sorties qui ne s'arrêtent pas correctement. Pour cette raison, la plupart des méthodes de l'état de l'art recourent à une stratégie de « fenêtre glissante », qui consiste à réordonner itérativement des sous-listes de taille réduite afin de couvrir l'ensemble des candidats. Cette stratégie vise avant tout à rendre la tâche plus fiable en diminuant la complexité du réordonnancement à chaque étape, et non à contourner une limite stricte de mémoire contextuelle. Elle reste toutefois coûteuse, car elle nécessite plusieurs inférences LLM successives pour une même requête, ce qui limite fortement les possibilités de parallélisation.

Dans cet article, nous proposons Filter and Rerank Efficiently with LLM (FIRE-LLM), une méthode qui combine filtrage et réordonnancement pour traiter efficacement de grandes listes candidates. FIRE-LLM insère un filtrage en amont du réordonnancement afin de réduire l'ensemble initial de passages à une taille raisonnable, avant de produire le classement final en une seule inférence. Le filtrage, également assuré par un LLM, vise à écarter les passages non pertinents parmi les candidats récupérés. Cette organisation améliore les résultats par rapport à un réordonnancement direct et atteint des performances proches des stratégies à fenêtre glissante de l'état de l'art. Enfin, FIRE-LLM réduit le temps d'inférence et la consommation énergétique par rapport à la fenêtre glissante, avec un gain qui s'accentue lorsque la taille de l'ensemble initial augmente.

La Figure 1 illustre la chaîne de traitement. La recherche initiale réduit le nombre de passages considérés en conservant les 100 (ou 200 ou 300) meilleurs candidats. Cet ensemble est filtré par le LLM pour obtenir une liste plus courte. Le LLM réordonne ensuite cette liste et produit le classement final.

Nous avons évalué FIRE-LLM sur deux collections de référence en RI, à savoir TREC Deep Learning Track 2019 et 2020 [2]. Nous avons comparé notre méthode à la stratégie à fenêtre glissante de l'état de l'art, ré-implémentée en utilisant le même LLM que FIRE-LLM, afin d'assurer une

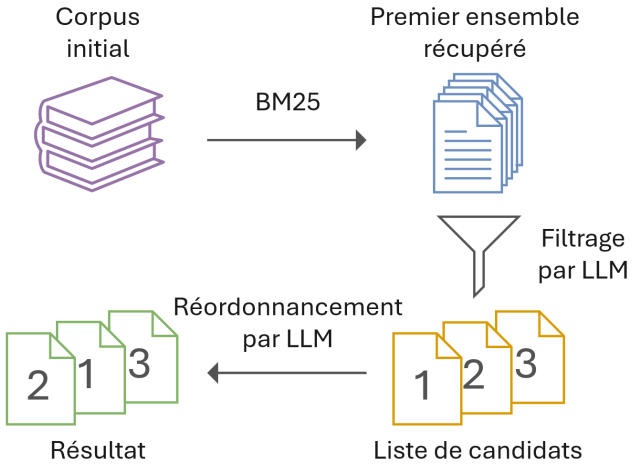


FIGURE 1 – Illustration de notre chaîne de traitement en trois étapes FIRE-LLM.

comparaison équitable. Cette ré-implémentation permet de mesurer le temps d'inférence, ainsi que d'estimer la consommation énergétique et les émissions de $CO_2$. Pour des performances comparables en NDCG@10, FIRE-LLM est 2 à 5 fois plus rapide et plus économe en énergie pour le réordonnancement de 100 passages. L'avantage augmente avec la taille de la liste candidate et atteint un facteur d'environ 14 en temps pour 300 passages.

**Définition de la tâche.** À partir d'une requête $q$ et d'un corpus de passages $P$, l'objectif est d'identifier un ensemble de passages candidats $S(q, P) \subseteq P$ et d'en produire un classement par pertinence pour $q$, noté $R(q, S) = (p_1, \ldots, p_n)$. Cette définition sépare la récupération de candidats et leur classement, ce qui correspond à l'architecture habituelle des systèmes de RI. Dans un contexte où des LLM sont utilisés, cette formulation permet en outre de contraindre la production d'information à des sources explicitement fournies, et d'éviter que le système ne s'appuie sur des connaissances issues des données d'entraînement du LLM. Elle s'applique notamment lorsque le corpus est interne, contrôlé, ou sensible (p.,ex. applications scientifiques et médicales, documents confidentiels).

La Section 2 positionne notre approche par rapport aux travaux antérieurs. La Section 3 décrit la méthodologie proposée et la Section 4 présente les expérimentations et l'analyse des résultats. Nous concluons en Section 5.

## 2 Travaux antérieurs

De nombreux travaux intègrent désormais des LLM dans les pipelines de recherche d'information (RI), notamment pour le réordonnancement. Deux axes sont particulièrement liés à notre problématique : (1) le *listwise reranking*, qui permet de comparer plusieurs passages conjointement ; (2) l'introduction d'un filtrage en amont afin de réduire le coût de calcul en limitant le réordonnancement à un sous-

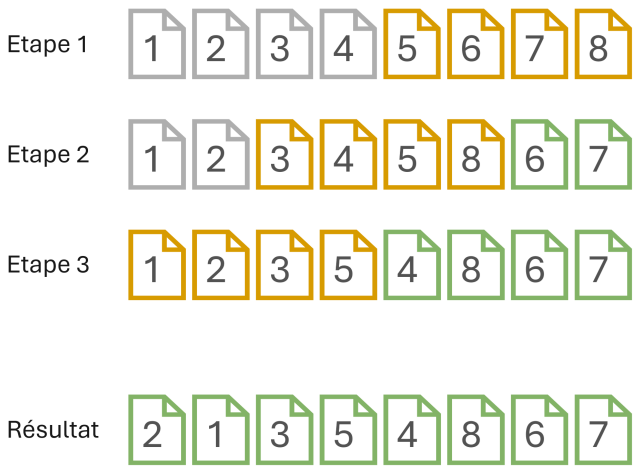


FIGURE 2 – Exemple de stratégie de fenêtre glissante : La fenêtre est indiquée en **orange** et les passages réordonnés en **vert**. Pour une fenêtre de taille 4 et un pas de 2, le réordonnancement d'une liste de 8 passages nécessite trois itérations.

ensemble de passages candidats. Cette section synthétise les contributions majeures de ces deux axes et situe notre approche par rapport à l'état de l'art.

## 2.1 *listwise reranking*

Le *listwise reranking* est aujourd'hui largement étudié [6, 10, 12]. Son intérêt est de permettre au LLM de considérer plusieurs passages conjointement et de comparer directement des candidats. Nous présentons ci-dessous deux travaux influents qui mettent en évidence ces avantages et ont motivé nos choix méthodologiques.

Sun et al. [12] proposent RankGPT, une approche de *listwise reranking* qui met en évidence l'influence du LLM sur les performances. Pour traiter des listes longues, RankGPT utilise une stratégie de « fenêtre glissante » : la liste est réordonnée itérativement de la fin vers le début à l'aide d'une fenêtre de taille fixée et d'un pas de glissement, de sorte que chaque inférence ne considère qu'un nombre limité de passages. Cette procédure permet de traiter un nombre arbitraire de passages récupérés, tout en contrôlant la charge imposée au LLM. Elle permet aussi d'éviter les problèmes liés au traitement simultané d'un trop grand nombre de passages, ou à un contexte trop grand pour la capacité du modèle.

La figure 2 illustre le réordonnancement de 8 passages avec une fenêtre glissante de taille 4 et un pas de 2. À chaque itération, le LLM traite les passages couverts par la fenêtre, puis la liste est mise à jour et la fenêtre avance de deux positions, jusqu'à couvrir l'ensemble des passages. Cette approche a deux principales limites. D'une part, les performances sont sensibles à l'ordre des passages dans le prompt, avec un écart notable entre le meilleur et le pire ordonnancement. D'autre part, la fenêtre glissante requiert plusieurs itérations pour traiter la liste complète, ce qui augmente le coût de calcul. Ce traitement est séquentiel au niveau d'une requête, ce qui limite la parallélisation et rend l'approche moins adaptée aux contextes exigeant de faibles temps de réponse.

Ma et al. [6] évaluent le *listwise reranking* avec LLM en contexte *zero-shot*. Ils obtiennent des résultats supérieurs par rapport au réordonnancement point par point, avec ou sans LLM. Leur étude compare un modèle pré-entraîné pour le *pointwise reranking*, ainsi que des approches *pointwise* et *listwise* utilisant GPT-3 comme LLM. Les expériences portent sur le réordonnancement des 100 premiers passages récupérés par BM25 sur plusieurs jeux de données. Pour traiter l'ensemble des passages dans le cadre *listwise*, les auteurs recourent à une stratégie de fenêtre glissante. Les résultats rapportés indiquent un avantage systématique de l'approche par liste sur les trois jeux de données considérés. Ils explorent ensuite un second protocole, où le *listwise reranking* est appliqué à une liste plus courte (10 ou 20 passages) issue d'une étape préalable réalisée par des méthodes *pointwise*. Cette combinaison (sélection préalable puis un unique passage *listwise*) améliore les performances par rapport aux baselines comparées.

Ces approches constituent une base importante pour notre travail. Elles mettent en évidence l'intérêt du *listwise reranking*, mais soulignent aussi le coût induit par la fenêtre glissante lorsque la liste candidate est longue. Notre contribution vise à réduire ce coût tout en maintenant une qualité de classement comparable.

## 2.2 Filtrer avant le réordonnancement

L'introduction d'un filtrage en amont du réordonnancement constitue une piste naturelle pour réduire le coût de calcul, mais elle a été relativement peu étudiée à ce jour. Nous présentons ci-dessous les travaux les plus représentatifs afin d'en préciser l'intérêt et les limites.

Meng et al. [7] comparent plusieurs stratégies de réduction de la liste candidate par filtrage avant l'étape de réordonnancement. Leurs résultats indiquent que les méthodes de filtrage supervisées n'apportent pas de gain par rapport à des méthodes non supervisées. De plus, une taille de liste fixée *a priori* conduit à des résultats proches de ceux obtenus lorsque la taille est optimisée dynamiquement. Ils observent également que le type de recherche initiale utilisé pour produire les passages candidats influence la taille de réduction optimale : une recherche initiale plus performante permet de réduire l'ensemble initial à des listes candidates plus petites tout en conservant un bon niveau de performance, ce qui améliore le compromis entre qualité et rapidité.

Nouriinanloo et al. [8] introduisent une étape de filtrage en amont du réordonnancement afin d'améliorer la qualité du classement. Ils montrent que ce filtrage réduit le nombre de faux positifs parmi les passages candidats, ce qui se traduit par une amélioration notable des performances. Leur méthode utilise un LLM pour attribuer à chaque passage un score de pertinence par rapport à la requête. Les passages dont le score est inférieur à un seuil sont ensuite écartés de la liste candidate. Le seuil est ajusté sur l'ensemble de validation propre à chaque jeu de données, via un processus

itératif visant à maximiser le score F1 des prédictions du LLM par rapport à la vérité terrain.
Dans la continuité de ces travaux, nous introduisons un filtrage préliminaire destiné à réduire la liste candidate avant réordonnancement. L'objectif est de diminuer le coût d'inférence de l'étape de réordonnancement, en réduisant le nombre de passages traités par le LLM, tout en maintenant des performances comparables.

# 3 Méthodologie

Dans la continuité de travaux récents [6, 12, 15], nous utilisons les LLM pour le réordonnancement tout en contrôlant le coût d'inférence. Nous proposons FIRE-LLM, une chaîne en trois étapes : (1) une recherche initiale rapide, (2) un filtrage par LLM pour réduire la liste candidate, puis (3) un réordonnancement en une passe sur la liste filtrée. L'objectif est de combiner l'efficacité de la recherche conventionnelle et les capacités des LLM à juger la pertinence des passages, afin d'obtenir un classement performant. Nous visons une qualité de classement comparable aux approches *listwise* de l'état de l'art, tout en réduisant significativement le nombre d'appels au LLM. Cette réduction diminue le temps d'inférence et, par conséquent, l'impact environnemental associé au processus.

## 3.1 Première étape de recherche rapide

Nous utilisons BM25 [11] comme première étape, car il s'agit d'une méthode largement adoptée pour la récupération initiale en recherche d'information. Cette étape doit satisfaire trois exigences pour assurer de bonnes performances en aval. Premièrement, elle doit être suffisamment rapide pour traiter de larges collections de passages et rester utilisable en pratique. Deuxièmement, elle doit présenter une bonne capacité de rappel, afin de limiter le risque d'écarter des passages pertinents dès la phase de récupération initiale. Enfin, elle doit produire une liste candidate de taille limitée, compatible avec les fenêtres de contexte des LLM modernes, afin de permettre un traitement ultérieur efficace. BM25 est fréquemment utilisé comme point de départ dans les chaînes de recherche d'information et constitue une référence standard pour comparer des méthodes de réordonnancement.

## 3.2 Étape de filtrage par LLM

La deuxième étape applique un filtrage par LLM à la liste candidate produite par BM25, afin de supprimer les passages non pertinents avant le réordonnancement. Ce filtrage sert de contrôle qualité : il réduit le bruit, améliore la précision et vise à conserver un rappel élevé. Le LLM attribue une estimation de pertinence à chaque passage vis-à-vis de la requête en une seule passe, permettant de filtrer efficacement l'ensemble des candidats.
Nous insérons dans la fenêtre de contexte du LLM l'ensemble des passages récupérés, représentés comme un ensemble de paires (identifiant, texte). Le prompt précise (i) la requête, (ii) la consigne de lecture et de comparaison, et (iii) la contrainte de produire une liste de taille fixe d'identifiants. Un format de sortie contrôlé permet d'extraire automatiquement la liste des identifiants et de récupérer les passages correspondants pour passer à l'étape finale. En complément, le LLM fournit un score de pertinence pour chaque identifiant sélectionné, qui définit un ordre préliminaire utilisé pour initialiser le réordonnancement.

Nous intégrons cette étape de filtrage entre la recherche initiale et le réordonnancement final afin d'éviter que le LLM utilisé pour le réordonnancement soit confronté à un contexte trop chargé en faux positifs. En effet, cette tâche est fortement sensible au nombre de passages fournis (i.e., à la longueur du contexte). À l'inverse, l'étape de filtrage tolère un nombre plus élevé de passages, car elle se limite à juger la pertinence de chaque passage indépendamment, sans devoir établir un ordre global.

## 3.3 Réordonnancement par LLM

La dernière étape produit le classement final via un *listwise reranking* en une seule passe sur la liste filtrée. Le LLM compare explicitement les passages au sein d'une même fenêtre de contexte, ce qui favorise des décisions de classement fondées sur l'ensemble des candidats plutôt que sur des évaluations indépendantes. Cette conception est rendue possible par l'étape de filtrage amont et permet de limiter le nombre d'appels au LLM, tout en conservant les bénéfices du paradigme *listwise*.

Nous fournissons au LLM la liste filtrée sous un format structuré associant à chaque passage un identifiant unique. La requête ainsi que les instructions sont ensuite ajoutés dans le même prompt. Les instructions demandent de réordonner les passages par pertinence vis-à-vis de la requête et de renvoyer uniquement la séquence d'identifiants correspondante. Grâce à ce format, la sortie peut être analysée automatiquement afin de reconstruire la liste finale réordonnée. La taille de cette liste dépend du nombre de passages conservés lors de l'étape de filtrage.

## 3.4 Ingénierie de prompt

La Figure 3 donne un exemple de prompt utilisé pour l'étape de filtrage, en distinguant le *system prompt* et le *user prompt*, ainsi que les champs injectés (`[INPUT ... ]`) et le paramètre contrôlant la taille de sortie (`<NUMBER>`). Pour l'étape de réordonnancement, nous gardons la même structure de prompt. Nous modifions les instructions pour reclasser l'intégralité de la liste fournie par ordre de pertinence. Pour une chaîne de traitements robuste, la sortie du modèle doit être structurée pour permettre une extraction automatique des identifiants. Dans ce but, nous avons construit ces prompts de manière à limiter les erreurs de format.

**System prompt :**

```
You are an assistant that can only use the
passages given in the user prompt. [...]
```

**User prompt :**

```
Below is a list of passages, each labeled
with a key:
[INPUT PASSAGES DICTIONARY]
Here is the web search query: [INPUT QUERY]
Instructions: Please identify the <NUMBER>
passages that best match the web search
query above. [...]
```

FIGURE 3 – Exemple de prompt pour l'étape de filtrage. Les entrées injectées dans le prompt sont indiquées par les balises `[INPUT ... ]`. Le paramètre `<NUMBER>` fixe le nombre de passages renvoyés en sortie.

# 4 Expérimentations et Analyse des résultats

Nous avons mené une série d'expérimentations pour évaluer l'efficacité de FIRE-LLM. Nous examinons d'abord les limites du réordonnancement en une seule passe appliqué à l'ensemble complet de passages issu de la recherche initiale, sans étape de filtrage ou stratégie de fenêtre glissante. Nous comparons ensuite FIRE-LLM à RankGPT, une approche de référence fondée sur le *listwise reranking*, afin d'évaluer la qualité de notre réordonnancement et nos gains en temps de calcul.
Les sous-sections suivantes présentent notre protocole expérimental ainsi qu'une analyse détaillée des résultats.

## 4.1 Configuration expérimentale

### 4.1.1 Jeux de données

Nous évaluons notre approche sur deux collections de référence pour la recherche de passages, issues de TREC Deep Learning Track 2019 et 2020 (TREC-DL-2019 et TREC-DL-2020) [2]. Toutes les expérimentations sont réalisées en configuration *zero-shot*, en utilisant les requêtes de test officielles associées à chaque édition, identiques à celles employées dans l'évaluation de RankGPT [12].

### 4.1.2 Modèles

Pour la récupération initiale, nous utilisons BM25 via `pyserini` [5] et retenons les $k$ premiers passages, avec $k \in 100, 200, 300$. `pyserini` fournit des index précalculés pour de nombreuses collections ainsi qu'une implémentation standardisée de BM25, ce qui facilite la reproductibilité et la comparaison avec des travaux antérieurs s'appuyant sur le même environnement.
Pour les étapes fondées sur un LLM, nous utilisons Qwen2.5-72B-Instruct [3]. Nous fixons ce modèle pour l'ensemble des méthodes évaluées. L'objectif est de comparer FIRE-LLM à une approche à fenêtre glissante, et non de comparer différents LLM. Ce choix se justifie par des considérations pratiques : le modèle est open source, s'exécute localement, dispose d'une fenêtre de contexte adaptée à nos prompts et permet de mesurer le temps d'inférence ainsi que d'estimer l'énergie et le $CO_2$ dans un environnement de calcul maîtrisé. Enfin, des essais préliminaires ont montré que Qwen2.5-72B-Instruct fournissait des performances suffisantes pour notre protocole. Nous retenons RankGPT [12] comme baseline principale car leur méthode constitue une référence du *listwise reranking* par LLM dans la littérature récente. Dans ce cadre, nous utilisons également Qwen2.5-72B-Instruct pour notre implémentation de référence de RankGPT (fenêtre glissante), de manière à évaluer RankGPT et FIRE-LLM dans des conditions identiques. La fenêtre glissante est paramétrée comme dans [12], avec une fenêtre $W = 20$ et un pas $S = 10$. Le LLM est exécuté en quantification 8 bits afin de réduire l'empreinte mémoire et d'améliorer la vitesse d'inférence. La configuration utilise une fenêtre de contexte maximale de 32 768 tokens et autorise des sorties jusqu'à 8 192 tokens.

### 4.1.3 Infrastructure de calcul

Toutes les expérimentations ont été réalisées sur un nœud de calcul comprenant deux GPU NVIDIA A100 (80 GB) et huit processeurs AMD EPYC 7543 (Milan).

## 4.2 Résultats

### 4.2.1 Performances

Pour l'évaluation, nous rapportons NDCG@10 sur TREC-DL-2019 et TREC-DL-2020. Nous comparons FIRE-LLM à BM25 comme base de référence, ainsi qu'à RankGPT [12] comme baseline *listwise* et à une approche naïve de réordonnancement direct. Le Tableau 1 regroupe, d'une part, des valeurs RankGPT issues de la littérature (exécutées avec `gpt-3.5-turbo` et `gpt-4`) et, d'autre part, les résultats obtenus dans nos propres conditions expérimentales. Les valeurs reportées depuis [12] sont fournies à titre indicatif et ne constituent pas une comparaison directe car elles reposent sur un LLM et un environnement d'exécution différents. Notre analyse se concentre donc sur la comparaison à LLM fixé, où toutes les approches utilisent Qwen2.5-72B-Instruct.
Nous évaluons d'abord la stratégie de réordonnancement direct en une seule passe sur la liste candidate issue de BM25. Pour $k \in \{100, 200, 300\}$, les passages récupérés sont injectés simultanément dans la fenêtre de contexte de Qwen2.5-72B-Instruct, qui doit renvoyer une liste classée d'identifiants. Comme le montre le Tableau 1, bien que cette approche surpasse BM25, elle reste en retrait par rapport à RankGPT ou FIRE-LLM. Nous constatons en outre une baisse de performance lorsque la liste candidate s'allonge ($k = 100$ vers 300), ce qui indique une forte sensibilité du modèle à gérer un trop grand nombre de passages en

TABLE 1 – Performances globales, évaluées à l'aide de la métrique NDCG@10 sur les jeux de données TREC-DL-2019 et TREC-DL-2020. Les résultats *RankGPT (littérature)* sont reportés depuis [12] à titre indicatif. La comparaison principale est réalisée à LLM fixé, avec Qwen2.5-72B-Instruct pour l'approche directe naïve, la ré-implémentation de la fenêtre glissante RankGPT et notre approche FIRE-LLM.

| **Méthodes** | | | **TREC-DL-2019** | **TREC-DL-2020** |
|---|---|---|---|---|
| **Modèle / Stratégie** | **BM25 $k$** | **Filtre $k$** | **NDCG@10** | **NDCG@10** |
| BM25 | 100 | – | 50.58 | 47.96 |
| *Références (RankGPT reporté dans la littérature)* | | | | |
| RankGPT (`gpt-3.5-turbo`) [12] | 100 | – | 65.80 | 62.91 |
| RankGPT (`gpt-4`) [12] | 100 | – | 75.59 | 70.56 |
| *Résultats obtenus dans nos conditions expérimentales (Qwen2.5-72B-Instruct)* | | | | |
| Direct Rerank (naïf) | 100 | – | 66.71 | 63.97 |
| RankGPT (fenêtre glissante) | 100 | – | 70.11 | 66.90 |
| FIRE-LLM | 100 | 20 | 68.26 | 66.20 |
| Direct Rerank (naïf) | 200 | – | 63.25 | 61.82 |
| RankGPT (fenêtre glissante) | 200 | – | 72.70 | 68.82 |
| FIRE-LLM | 200 | 20 | 68.45 | 67.72 |
| FIRE-LLM | 200 | 30 | 70.52 | 66.71 |
| Direct Rerank (naïf) | 300 | – | 64.11 | 56.09 |
| RankGPT (fenêtre glissante) | 300 | – | 72.56 | 70.08 |
| FIRE-LLM | 300 | 30 | 70.54 | 65.50 |

une seule passe.

À l'inverse, dans ce cadre contrôlé, FIRE-LLM obtient des performances proches de RankGPT pour les deux jeux de données, quel que soit le nombre de passages récupérés par BM25. Avec $k = 100$, l'écart reste limité (70.11 vs. 68.26 sur TREC-DL-2019 ; 66.90 vs. 66.20 sur TREC-DL-2020), tout en ne nécessitant que deux appels au LLM. Lorsque $k$ augmente, FIRE-LLM conserve des résultats compétitifs, avec des variations liées au seuil de filtrage (par exemple, $k = 200$ avec un filtre à 30 passages améliore NDCG@10 sur TREC-DL-2019). Ces résultats indiquent que la réduction préalable de la liste candidate permet de se rapprocher des performances d'une stratégie à fenêtre glissante, tout en limitant le nombre d'inférences et, par conséquent, le coût d'inférence.

#### 4.2.2 Fiabilité du réordonnancement direct

Afin d'investiguer l'origine de la contre-performance de l'approche directe (mise en évidence dans le Tableau 1), nous en caractérisons la fiabilité en examinant les listes générées. Les métriques du Tableau 2 confirment cette dégradation (e.g. identifiants invalides ou répétés, temps d'inférence en hausse) induite par cette approche naïve. Ces paramètres mesurent la capacité du LLM à produire une sortie conforme au format attendu lorsque la liste candidate est longue :

- **Longueur moyenne des listes générées** : longueur moyenne des listes produites. Un écart à $k$ indique une sortie tronquée (ou incomplète) par rapport à la liste candidate BM25.
- **Générations sans fin** : proportion de requêtes pour lesquelles le modèle ne termine pas sa génération et ne produit pas de sortie valide. Ce phénomène est généralement associé à des prompts trop longs ou à une difficulté à suivre le format de réponse demandé.
- **Nombre moyen d'identifiants invalides par liste** : nombre moyen d'identifiants produits qui ne correspondent à aucun passage fourni (hors plage) ou qui ne sont pas interprétables comme identifiants (e.g. chaînes non numériques). La présence de tels éléments indique que le modèle ne respecte pas l'espace d'identifiants imposé, ce qui dégrade la validité de la sortie.
- **Nombre moyen d'identifiants répétés par liste** : nombre moyen d'identifiants répétés dans les listes produites. Ces répétitions signalent une dégradation de la conformité de la sortie, puisqu'un identifiant ne doit apparaître qu'une seule fois.
- **Temps moyen d'inférence par requête** : durée moyenne de génération par requête ; cette mesure reflète directement le coût de calcul associé au réordonnancement.

Ces métriques sont récapitulées dans le Tableau 2. Les résultats indiquent une dégradation de la conformité des sorties lorsque la liste candidate dépasse 100 passages : le nombre d'identifiants invalides et répétés augmente nettement, de même que la proportion de requêtes conduisant à des générations sans fin. La longueur moyenne des listes générées met également en évidence un écart systématique comparativement à la taille attendue, les sorties contenant environ la moitié des passages fournis quelle que soit la valeur de $k$. Enfin, le temps d'inférence croît fortement avec la taille de la liste candidate, ce qui limite l'intérêt pratique de cette stratégie.

TABLE 2 – Métriques de fiabilité issues du réordonnancement direct de l'ensemble des passages récupérés par BM25, sur le jeu de données TREC-DL-2020.

| **Métriques** | **Runs** | | |
|---|---|---|---|
| **BM25 k** | **100** | **200** | **300** |
| Longueur moyenne des listes générées | 49.81 | 97.91 | 160.67 |
| Générations sans fin | 0% | 3.70% | 9.26% |
| Nombre moyen d'identifiants invalides par liste | 0.00 | 2.50 | 11.78 |
| Nombre moyen d'identifiants répétés par liste | 0.24 | 1.43 | 3.22 |
| Temps moyen d'inférence par requête | 73s | 220s | 439s |

#### 4.2.3 Temps et consommation énergétique

Nous mesurons le coût d'inférence des étapes par LLM en termes de temps (s), d'énergie (kWh) et d'émissions de $CO_2$ (g). Les mesures sont effectuées sur l'ensemble des requêtes puis moyennées par requête. Comme RankGPT ne fournit pas de mesures de temps, d'énergie ou de $CO_2$, nous utilisons notre réimplémentation de RankGPT (décrite précédemment) et l'exécutons dans les mêmes conditions que FIRE-LLM, avec le même LLM, afin de garantir une comparaison équitable. Les estimations d'énergie et de $CO_2$ sont obtenues avec `CodeCarbon` [4], à partir de la consommation mesurée sur la machine et d'une intensité carbone dépendante du lieu et du moment. Toutes les expériences sont réalisées sur la même infrastructure, située en France.

Les résultats de la Figure 4 indiquent que FIRE-LLM diminue le coût d'inférence par requête relativement à RankGPT, tant en temps qu'en énergie et en émissions de $CO_2$. Cette différence provient de la structure de FIRE-LLM, qui effectue un filtrage puis un réordonnancement en une passe sur la liste filtrée, soit un nombre constant d'appels au LLM pour une taille initiale donnée. À l'inverse, RankGPT traite la liste candidate via une fenêtre glissante et requiert plusieurs passes séquentielles. Le nombre de passes associé à RankGPT est donné par la formule suivante :

$$\text{Nombre de passes} = \left\lceil \frac{\mathrm{P} - (\mathrm{W} - \mathrm{S})}{\mathrm{S}} \right\rceil \qquad (1)$$

où $P$ désigne le nombre de passages à ordonner, $W$ la taille de la fenêtre, et $S$ le pas de glissement. Par exemple, en utilisant la configuration principale de RankGPT décrite dans [12] avec $P$=100, $W$=20 et $S$=10, le nombre de passes s'élève à 9. Même avec $P = 100$, le gain est déjà net en temps de traitement et en émissions de $CO_2$ : FIRE-LLM est environ deux à cinq fois plus rapide et émet environ deux à cinq fois moins de $CO_2$. Lorsque $P$ augmente (200, 300 ou davantage), l'écart se creuse et FIRE-LLM peut être environ quatorze fois plus rapide, en raison du nombre de passes imposé par la fenêtre glissante. À notre connaissance, il n'existe pas de mesures publiques directement comparables (temps, énergie, $CO_2$) pour `gpt-3.5-turbo` et `gpt-4` dans des conditions expérimentales alignées avec les nôtres. Nous rapportons donc des mesures obtenues sur notre infrastructure, en comparant FIRE-LLM à une implémentation de référence de RankGPT exécutée avec le même LLM, afin d'évaluer les stratégies dans des conditions identiques.

## 5 Conclusion

Dans cet article, nous avons présenté FIRE-LLM, une chaîne de réordonnancement structurée autour de trois étapes : récupération initiale par BM25, filtrage par LLM, puis réordonnancement final par LLM. Le filtrage contrôle la taille de la liste candidate et permet de produire le classement final en une passe. Les résultats expérimentaux indiquent des performances comparables aux approches par fenêtre glissante, pour un coût d'inférence sensiblement plus faible en temps et en énergie. Nos principaux résultats se déclinent en trois points. Premièrement, nous montrons que le réordonnancement direct en une seule passe sur des listes candidates longues dégrade la qualité du classement et entraîne des sorties moins fiables, ce qui motive l'introduction d'un mécanisme de réduction en amont. Deuxièmement, nous montrons que le filtrage proposé permet de ramener la liste candidate à une taille compatible avec un *listwise reranking* en une seule inférence, évitant ainsi le recours à une fenêtre glissante. Troisièmement, nos résultats expérimentaux indiquent que FIRE-LLM atteint des performances comparables à celles de RankGPT, tout en ne nécessitant que deux appels au LLM. Cette réduction du nombre de passes se traduit par une baisse nette du temps d'exécution et des émissions de $CO_2$. Enfin, l'écart d'efficacité augmente avec la taille de la liste candidate, ce qui rend l'approche pertinente lorsque le nombre de passages à traiter est élevé.

Cependant, FIRE-LLM présente plusieurs limites. D'une part, les performances dépendent du LLM utilisé, et peuvent varier selon le modèle et sa configuration. Cette dépendance est particulièrement marquée au niveau du filtrage, qui sollicite fortement la fenêtre de contexte et la capacité du modèle à évaluer un grand nombre de passages. En outre, toute erreur de filtrage peut se répercuter sur l'étape de réordonnancement en écartant des passages pertinents. D'autre part, la chaîne repose sur deux inférences successives (filtrage puis réordonnancement) qui sont séquentielles au niveau d'une requête : même si le trai-

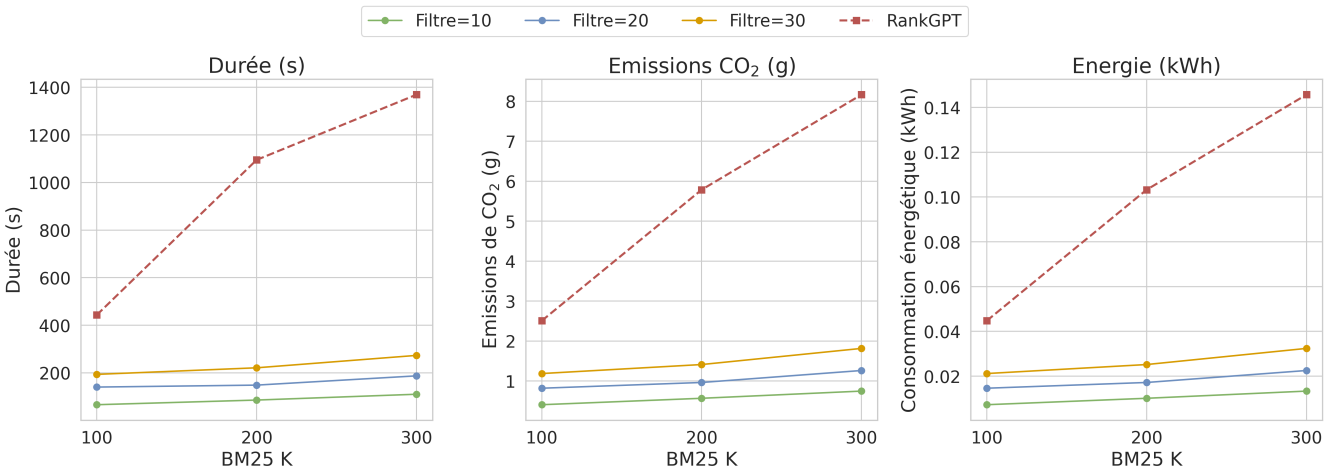


FIGURE 4 – Temps moyen d'inférence, émissions de $CO_2$ et consommation énergétique par requête. Nous comparons FIRE-LLM (pour différents seuils de filtrage) à notre réimplémentation de RankGPT, toutes deux exécutées avec Qwen2.5-72B-Instruct.

tement peut être parallélisé sur plusieurs requêtes, ce caractère séquentiel limite le débit et la latence atteignables. Enfin, le temps d'inférence demeure sensible au volume de texte traité et augmente avec le nombre de passages fournis au LLM, ce qui impose un réglage attentif de la taille de la liste candidate pour des scénarios soumis à de fortes contraintes de temps réel.

Enfin, nous considérons comme travail futur l'intégration de mécanismes de cache pour limiter les calculs redondants. Nous nous intéressons en particulier au cache Clé–Valeur des LLM et aux stratégies de réutilisation proposées récemment [14, 13]. Dans nos expérimentations, nous observons que certains passages sont fréquemment réutilisés d'une requête à l'autre, ce qui rend coûteux leur traitement répété. La réutilisation de caches associés à des passages déjà traités pourrait ainsi réduire la latence et le coût d'inférence des étapes de filtrage et de réordonnancement et améliorer la viabilité de telles chaînes en conditions de déploiement.

## Disponibilité du code

Une version améliorée et plus récente du code source qui a permis de mener toutes ces expérimentations est disponible à l'adresse suivante : https://gite.lirmm.fr/lremy/fire-llm.

## Usage de l'IA générative

L'usage de l'IA générative dans le cadre de la rédaction de cet article se limite seulement à la révision orthographique ainsi qu'à une aide à la traduction de certains termes. Nous garantissons son non-usage pour toutes les données, tableaux et figures présentés dans cet article.

## Remerciements

Ces travaux ont bénéficié d'un accès aux ressources de calcul en HPC/IA et de stockage au IDRIS au travers de l'allocation de ressources 20XX-AD011014995R1 attribuée par GENCI sur la partition A100 du calculateur Jean Zay.

## Références

[1] Shijie Chen, Bernal Jimenez Gutierrez, and Yu Su. Attention in large language models yields efficient zero-shot re-rankers. In *The Thirteenth International Conference on Learning Representations (ICLR 2025)*, 2025.

[2] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, and Daniel Campos. Overview of the TREC 2020 deep learning track. In *Proceedings of the Twenty-Ninth Text REtrieval Conference (TREC 2020)*, 2020.

[3] An Yang et al. Qwen2.5 technical report. (arXiv :2412.15115), 2024.

[4] Benoit Courty et al. mlco2/codecarbon : v3.0.1, 2025. https ://doi.org/10.5281/zenodo.15320006.

[5] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. Pyserini : A Python toolkit for reproducible information retrieval research with sparse and dense representations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*, 2021.

[6] Xueguang Ma, Xinyu Zhang, Ronak Pradeep, and Jimmy Lin. Zero-Shot Listwise Document Reranking with a Large Language Model. (arXiv :2305.02156), 2023.

[7] Chuan Meng, Negar Arabzadeh, Arian Askari, Mohammad Aliannejadi, and Maarten de Rijke. Ranked List Truncation for Large Language Model-based Re-Ranking. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2024)*, 2024.

[8] Baharan Nouriinanloo and Maxime Lamothe. Re-Ranking Step by Step : Investigating Pre-Filtering for Re-Ranking with Large Language Models. (arXiv :2406.18740), 2024.

[9] Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Le Yan, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, Xuanhui Wang, and Michael Bendersky. Large language models are effective text rankers with pairwise ranking prompting. In *Findings of the Association for Computational Linguistics : (NAACL 2024)*, 2024.

[10] Mandeep Rathee, Sean MacAvaney, and Avishek Anand. Guiding retrieval using llm-based listwise rankers. In *Advances in Information Retrieval : 47th European Conference on Information Retrieval, (ECIR 2025)*, 2025.

[11] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework : Bm25 and beyond. *Foundations and Trends in Information Retrieval*, 2009.

[12] Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. Is ChatGPT Good at Search ? Investigating Large Language Models as Re-Ranking Agents. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 2025)*, 2023.

[13] Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. Kvlink : Accelerating large language models via efficient kv cache reuse. (arXiv :2502.16002), 2025.

[14] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend : Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys 2025)*, 2025.

[15] Yutao Zhu, Huaying Yuan, Shuting Wang, Jiongnan Liu, Wenhan Liu, Chenlong Deng, Haonan Chen, Zheng Liu, Zhicheng Dou, and Ji-Rong Wen. Large language models for information retrieval : A survey. (arXiv :2308.07107), 2024.

[16] Honglei Zhuang, Zhen Qin, Kai Hui, Junru Wu, Le Yan, Xuanhui Wang, and Michael Bendersky. Beyond yes and no : Improving zero-shot LLM rankers via scoring fine-grained relevance labels. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies (NAACL 2024)*, 2024.

[17] Shengyao Zhuang, Honglei Zhuang, Bevan Koopman, and Guido Zuccon. A setwise approach for effective and highly efficient zero-shot ranking with large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2024)*, 2024.