A Survey on Sorting with Large Language Models

Ryoma Sato *rsato@nii.ac.jp*
National Institute of Informatics

Data Processing Club Technical Report DPC-TR-2026-001
February 10, 2026

Abstract

Sorting is a classical task in computer science, yet it has recently converged with state-of-the-art LLMs, giving rise to a new research trend. Sorting can leverage existing algorithms as long as a comparison function is defined. Traditional comparison functions typically assumed measurable numerical values such as height, price, or distance. With LLMs, however, it becomes possible to compare ambiguous and subjective concepts such as "which is preferred," "which is more persuasive," or "which is more relevant to the query." By invoking an LLM within the comparison function, sorting based on these concepts becomes feasible. Since the cost model for LLM-based comparisons differs from that of traditional numerical comparisons, new algorithm designs may be required. This paper surveys the foundations of LLM sorting, categorizes design methodologies, and reviews application examples.

1 Introduction

1.1 Sorting Can Be Generalized Around the "Comparison Function"

Sorting is a fundamental concept in computer science, appearing as a core component in many systems—information retrieval, recommendation, visualization, decision-making—where the central question is "what to present at the top." Classical sorting typically assumed measurable numerical values such as height, price, or distance. In this setting, sorting algorithms (e.g., quicksort) invoke a "comparison function that takes two elements and returns which one ranks higher" as a black box, and it is well known that the number of comparisons can be kept to $O(n \log n)$.

With LLMs, the comparison function itself can be implemented in natural language. That is, ambiguous and subjective concepts such as "which is preferred," "which is more persuasive," or "which is more relevant to the query" can become the basis for comparison. In recent years, classical sorting techniques have converged with state-of-the-art LLMs, giving rise to a new research trend. This paper surveys recent research trends from the perspective of "sorting with LLMs."

1.2 Motivating Example: Sorting Papers by Preference and News by Optimism

To build intuition for LLM sorting, we illustrate it through the example of paper curation. Human preferences such as "connoisseur taste" or "papers that answer 'why"' are difficult to quantify, but by providing comparison criteria to an LLM in natural language, one can create a comparison function that returns which of two papers better matches the criteria. For example, the following prompt can be considered.

```
I have been researching machine learning for about 10 years. I want to read papers
with connoisseur taste.
- I like papers that answer "why"
- I prefer technically non-trivial ideas over straightforward combinations of known
techniques
- I like general-purpose techniques that can be useful over a long period
```

```
Papers I especially liked among what I have read:
- Adversarial Examples Are Not Bugs, They Are Features
- Physics of Language Models: Part 2.1, Grade-School Math and the Hidden Reasoning
Process
- The Platonic Representation Hypothesis
- Language Models Learn to Mislead Humans via RLHF
- Arithmetic Without Algorithms: Language Models Solve Math With a Bag of Heuristics

Compare the following two papers and select the one that better matches the above
preferences.
[Paper A] Title: ... / Abstract: ...
[Paper B] Title: ... / Abstract: ...

Answer must be exactly "A" or "B" only. No explanation needed.
```

By plugging an LLM call using this prompt into an existing sort, paper candidates can be reordered according to preference.

```python
import functools

def llmcompare(paper_a, paper_b):
    prompt = build_prompt(paper_a, paper_b)
    ans = client.responses.create(
        model="gpt-5.2",
        input=prompt,
    ).output_text.strip().upper()   # "A" or "B"
    return -1 if ans.strip() == "A" else 1

papers_sorted = sorted(papers, key=functools.cmp_to_key(llmcompare))
```

Listing 1: Plugging an LLM comparison function into an existing sort (conceptual example)

When I sorted the oral papers accepted at ICLR 2026 with this, I obtained the following results.

```
How Do Transformers Learn to Associate Tokens: Gradient Leading Terms Bring
Mechanistic Interpretability
Verifying Chain-of-Thought Reasoning via its Computational Graph
The Spacetime of Diffusion Models: An Information Geometry Perspective
From Markov to Laplace: How Mamba In-Context Learns Markov Chains
Energy-Based Transformers are Scalable Learners and Thinkers
The Shape of Adversarial Influence: Characterizing LLM Latent Spaces with Persistent
Homology
Temporal superposition and feature geometry of RNNs under memory demands
Sequences of Logits Reveal the Low Rank Structure of Language Models
Distributional Equivalence in Linear Non-Gaussian Latent-Variable Cyclic Causal Models
Reasoning without Training: Your Base Model is Smarter Than You Think
.
.
.
```

Papers that match my taste are selected at the top. This should make it easier to decide what to read next.

If you implement the comparison function with a prompt like “Compare the following two economic news items and decide which one is more optimistic,” and then sort recent economic news headlines from the most optimistic to the most pessimistic, you get the following results. Here are the top 10 and bottom 10.

```
Inflation: Japan improves. Europe sees a gradual recovery, and the US sees a strong
recovery. Stock and metal markets hit all-time highs.
Dow Jones closes above 50,000 for the first time, on optimistic views about the US
economic outlook
Dow 50,000: AI changes the drivers; Caterpillar stands out with stock price doubling
Nikkei average hits record high, closing up 2,065 yen at 54,720 yen; semiconductor
rally reignites
In 2026, Japanese stocks are expected to be driven by a turn to positive real wage
growth and "higher ROE"; a "shift into Japanese stocks" by retail investors is also
anticipated
Dow surpasses 50,000 for the first time; leadership shifts from tech to manufacturing
and resources
Dow rises 515 points; manufacturing sentiment far exceeds market expectations
Google's parent posts quarterly results with 30% profit increase; AI-related business
is strong
Amazon posts quarterly results with higher revenue and profit; core businesses such as
 cloud are strong
TSMC to produce advanced 3nm semiconductors in Kumamoto; performance and use cases are
.
.
.
Russia's GDP growth slows sharply: from over 4% to +1.0% in 2025; domestic demand
stagnates and inflation hits 9% amid the invasion of Ukraine
Engel coefficient in 2025 hits highest level in 44 years; real consumer spending in
December down 2.6%
Bitcoin is on the brink of extinction... why a long "winter" is expected
China's growth rate is "actually far below 5%": Wu Junhua
President Zelensky condemns Russian attacks: "They prioritize only war"
China rushes "war preparations" shock; deflation acceleration is unavoidable
Japan's financial crisis, almost identical... three risks of worsening US-China
relations that could hit China's cliff-edge economy bloated with debt
China that Japan must know now: China's economy faces risk of entering a deflationary
spiral; weak domestic demand and population decline also weigh
"There is no exit..." the decisive reason China's deflation is more severe than Japan'
s "lost 30 years"
[China's economy in 2026] A time bomb toward collapse... a crisis where even SOEs are
abandoned; a wave of real-estate and construction firm failures leading to a financial
 crisis
```

When sorting 295 news headlines, the sort performed 2,324 comparisons and finished in about 2.5 minutes. Using GPT-5-mini (with thinking enabled) cost \$0.85. Using GPT-5-nano reduces the cost to about one fifth.

Beyond this, you can sort politicians from left to right, sort bug reports by severity, and more. Even for tasks that require subjective and vague judgments, you can easily sort by using an LLM.

This plug-and-play nature is a key strength of LLM sorting, but there are also important caveats:

- **Non-transitivity / incomplete ordering**: Cycles can arise where $A \succ B$ and $B \succ C$, yet $C \succ A$.
- **Positional bias**: Judgments can change depending on the input order (the order in which A/B are presented) [53].
- **Cost**: Since the LLM is invoked for each comparison, the number of calls and token volume become the primary bottlenecks.

This paper surveys methodologies for performing LLM sorting practically while addressing these challenges.

2 Foundations: From Comparisons to Rankings

2.1 Fundamentals of Classical Sorting Algorithms

In this paper, we consider "descending sort," which places the superior items first.

Bubble sort compares the n-th item with the $(n-1)$-th item and swaps them if they are in the wrong order. Then it compares the $(n-1)$-th item with the $(n-2)$-th item and swaps them if they are in the wrong order. This operation is repeated down to the 1st item. As a result, the best item moves to the 1st position. The same operation is then repeated starting again from the n-th item, moving the second-best item to the 2nd position. By repeating this operation $n-1$ times, the entire list is sorted. The number of comparisons in bubble sort is $\frac{n(n-1)}{2}$.

Quicksort selects a pivot element as a reference and partitions the other elements into a group of those superior to the pivot and a group of those inferior to the pivot. The same operation is then applied recursively to each group. Finally, the groups are concatenated to produce a fully sorted list. If the pivot is chosen as the median, sorting can be done in $O(n \log n)$ comparisons even in the worst case. If the pivot is chosen randomly, the expected number of comparisons is $O(n \log n)$.

Insertion sort takes unsorted items one by one, appends each to the end of the sorted list, and then—similarly to bubble sort—repeatedly swaps adjacent elements from the end until the item is in its correct position. Starting from an empty list, this is repeated until all items have been inserted into the sorted list. The number of comparisons in insertion sort is $O(n^2)$. However, whereas bubble sort always requires $\frac{n(n-1)}{2}$ comparisons, insertion sort can achieve as few as $O(n)$ comparisons in the best case when the items are already roughly sorted. For this reason, insertion sort is effective when a rough ordering is available from BM25, rule-based methods, or similar approaches.

Heapsort is a sorting algorithm based on the heap data structure. Conceptually, a heap is a complete binary tree in which each parent node is inferior to its child nodes. First, each item is added to the bottom of the heap, and the heap property is maintained by comparing with the parent and swapping as necessary. Next, the root node (the least superior item) is extracted and placed at the end of the array, thereby determining the position of the least superior item. After this, the last element in the heap is moved to the root, and the heap property is restored by comparing with child nodes and swapping as necessary. This operation is repeated until all items have been extracted. The number of comparisons in heapsort is $O(n \log n)$ in the worst case.

2.2 Problem Setup: The "Comparison Function" via LLMs

Consider a binary relation $\succ$ defined over n items (documents, responses, models, etc.) $\{x_i\}_{i=1}^n$. When defining the binary relation using an LLM, the typical approach is to use a pairwise comparison prompt to ask the LLM whether $x_i \succ x_j$. In this case, the LLM often returns which is better in the form A/B.

However, the typical conditions for comparison sort to function correctly are:

- **Antisymmetry**: If $x_i \succ x_j$ and $x_j \succ x_i$, then $x_i = x_j$.
- **Transitivity**: If $x_i \succ x_j$ and $x_j \succ x_k$, then $x_i \succ x_k$.
- **Completeness**: Either $x_i \succ x_j$ or $x_j \succ x_i$ holds.

LLM comparisons can violate antisymmetry and transitivity due to positional bias (dependence on input order) and lack of consistency. Moreover, LLM responses are stochastic, and different results may be obtained even for the same pairwise comparison. Therefore, LLM sorting is not sorting in the strict sense; it is often a "pseudo-sort," and the resulting ranking depends on the algorithm (i.e., the order in which comparisons are made).

Much research has been conducted on addressing this discrepancy.

Stochasticity can be addressed by using greedy decoding. When stochastic fluctuations are unavoidable due to API constraints or other factors, results can be cached for pairs that have already been compared, avoiding redundant LLM calls for the same pair. This not only makes the binary relation deterministic within the algorithm, but also has the benefit of reducing the number of LLM calls [50]. Additionally, there are methods that account for stochasticity by generating multiple responses and determining superiority by majority vote [20, 24, 26, 44, 46, 52].

Regarding antisymmetry and completeness, the positional bias of LLMs poses a problem. When an LLM performs a comparison, it may be biased toward selecting the option presented first [21, 42, 53]. For example, Zheng et al. [53] asked GPT-4 which was better—a response generated by GPT-3.5 or one by Vicuna-13B. When the GPT-3.5 response was presented first, GPT-4 chose GPT-3.5; when Vicuna-13B was presented first, GPT-4 chose Vicuna-13B. In other words, GPT-4's answers were inconsistent, and in this case, it was impossible to determine which was truly better. If one naively uses an LLM as a comparison function without accounting for this, elements that happen to be presented first more often may be unfairly favored. To mitigate this bias, a method has been proposed where both $\texttt{llmcompare}(x_i, x_j)$ and $\texttt{llmcompare}(x_j, x_i)$ are called: if x_i is consistently chosen, then $x_i \succ x_j$; if x_j is consistently chosen, then $x_j \succ x_i$; if the results disagree, it is treated as a tie [34, 53]. In the case of a tie, one can either use a sorting algorithm that handles ties or fall back to a different criterion.

The lack of transitivity corresponds theoretically to the problem of Aggregating Inconsistent Information [4]. The feedback arc set problem on tournaments is the problem of finding a ranking of n elements that agrees with the largest number of pairwise comparison results, given $\binom{n}{2}$ pairwise comparisons among them. These comparison results need not be transitive; when they are not, the goal is to find a ranking that minimizes the number of contradicted pairwise comparisons. This problem is NP-hard, but approximation algorithms and heuristics have been proposed. For example, KwikSort [4], similarly to quicksort, selects a random pivot element and partitions the other elements into two groups based solely on their comparison results with the pivot, then sorts each group recursively. This yields a 3-approximation solution efficiently. This technique can also be applied in LLM sorting to estimate stable rankings from non-transitive comparison results. Additionally, even without theoretical guarantees, simple sorting algorithms such as heapsort can sometimes be effective in LLM sorting [34].

So far, we have considered the problem of complete sorting, but in practice, a total order is not always necessary. For example, in search, the top-k results are what matter most. In such cases, "partial sorting" rather than "complete sorting" may be performed. For instance, bubble sort can achieve complete sorting in $(n-1)$ passes, but by executing only k passes, the top-k items are guaranteed to be correctly sorted. This allows the task to be accomplished with fewer comparisons than a complete sort [34].

3 Taxonomy: Pointwise / Pairwise / Listwise / Setwise

Beyond the plug-and-play approach, there exist other methods for sorting with LLMs.

3.1 Listwise: Reordering in a Single Call

The most straightforward approach is to input all items to be sorted at once and have the LLM output their ordering. This is called the listwise approach. For example, when ranking by relevance to a query, the following prompt can be considered [43].

```
This is RankGPT, an intelligent assistant that can rank passages based on their
relevancy to the query.
The following are {{num}} passages, each indicated by number identifier []. I can rank
 them based on their relevance to query: {{query}}
[1] {{passage_1}}
[2] {{passage_2}}
...
```

```
[8] {{passage_8}}
The search query is: {{query}}
I will rank the {{num}} passages above based on their relevance to the search query.
The passages
will be listed in descending order using identifiers, and the most relevant passages
should be listed
first, and the output format should be [] > [] > etc, e.g., [1] > [2] > etc.
The ranking results of the {{num}} passages (only identifiers) is:
```

The LLM is prompted to generate the continuation. For example, the output might be:

```
[8] > [3] > [1] > [5] > [2] > [4] > [6] > [7]
```

This output can be directly used as the reordering result.

Advantages

- Requires fewer calls.
- Since multiple candidates are viewed simultaneously within the same prompt, more information can be leveraged for context-based comparisons.

Disadvantages

- Stabilizing the output is difficult.
- Input length constraints: In particular, as the number of items increase, the input and output grow correspondingly, making the task per call more difficult and prone to failure.
- Positional bias exists, making results unstable and dependent on input order.

In the listwise approach, the ordering is output in a fixed format, but ensuring the LLM outputs in this exact format is challenging. For example, omission of some items, duplication, or output in an unparseable format can occur [43]. For local models, this can be addressed with constrained decoding, but for API-based LLMs, this is difficult. In such cases, retries must be performed until the constraints are satisfied, increasing cost.

In practice, when the number of candidate items is large, it is not possible to input all items at once, so approximate processing is commonly performed. The most widely used method is the sliding window approach. First, a list of items sorted by a simple criterion such as a rule-based method is prepared. The bottom w items are input to the LLM for reordering. The window then slides up by $w/2$ items, and reordering is performed again. This operation is repeated until the top of the list is reached. This allows items to be reordered with $O(n/w)$ LLM calls [28, 43]. Although this is an incomplete sort, it can surface items that were missed by the classical criteria through LLM-based comparison, while avoiding the $O(n \log n)$ comparisons of exact sorting—making it a popular compromise. In particular, for tasks such as search result reranking where exact sorting is not necessary and one simply wants to refine the top results, this approach is effective. However, it should be noted that this is an incomplete sort. In particular, this method makes it difficult for items initially ranked highly to drop significantly in rank. An item initially ranked 1st can only drop to rank w at worst. Therefore, if the initial ranking is poor and there are candidates that "must absolutely not appear" near the top, the window width should be at least as large as the size of the final ranking to be used, or a complete sort should be performed through a different mechanism.

3.2 Pointwise: Scoring Items and Sorting Conventionally

At the opposite extreme lies the pointwise approach. The pointwise method inputs each item independently and obtains a score for it. These scores are then used as keys to run a conventional sorting algorithm. For example, the following prompt can be considered:

```
Please evaluate how well the following paper matches my preference criteria on a scale
 of 1 to 5.
[Paper A] Title: ... / Abstract: ...

Score (1-5):
```

The LLM is prompted to generate the continuation. For example, the output might be:

```
4
```

However, outputting only scores from 1 to 5 results in a large number of ties. To avoid this, a method has been proposed that weights scores by the probability of outputting each digit token [19, 24, 47]

$$\text{score}(x) = \sum_{i=1}^{5} i \times P(\text{LLM outputs “}i\text{”} \mid x) \tag{1}$$

This allows for nuanced scoring that captures subtleties even when the LLM is uncertain between, say, a score of 3 and 4.

Advantages

- Only n calls are needed.
- Less susceptible to positional bias.
- Less susceptible to bias toward length or assertive tone [17, 45].

Disadvantages

- Designing an absolute scale is difficult.
- Only a single item is input at a time, limiting the available information.

In the pointwise approach, scores are assigned without seeing other items. Ideally, when the quality of competitors is low, one could afford to be lenient, and when the overall quality of competitors is high, one should be stricter to differentiate effectively. However, assigning scores independently prevents such calibration. The scoring criteria may drift between calls, causing items that happen to be scored when the LLM is in a “generous mood” to be ranked higher than they deserve.

3.3 Pairwise: Comparisons Are What LLMs Excel At

The plug-and-play approach described in Section 2 is classified as pairwise. In the pairwise approach, x_i and x_j are presented together, and the LLM is asked to choose “which is better.” This approach has the following advantages and disadvantages.

Table 1: Taxonomy of LLM sorting methods (representative examples)

Category	Typical Output	# Calls	Representative Examples
pointwise	Score (e.g., 1–5)	$O(n)$	G-Eval [24]
pairwise	A/B/Tie	$O(n \log n)$ / $O(n^2)$	PRP [34]
listwise	ID sequence	$O(1)$ / $O(n/w)$ (window w)	RankGPT family [43], LRL [28], RankVicuna/Zephyr [32, 33]
setwise	Top ID / Insertion position	$O(n \log_c n)$ etc.	Setwise prompting [55], Setwise insertion [31]

Advantages

- Comparison is easier than absolute evaluation or reordering everything at once. Even for humans, comparing two items is easier than assigning absolute scores to items or reordering everything at once. Similarly, LLMs find relative comparisons easier and more stable. This is especially pronounced when the model is small and has lower capability [34].
- Existing sorting algorithms can be leveraged. As we will see later, beyond simple sorting, there are various algorithms and analytical results for parallelizable sorting or sorting when cheap comparisons are available, and these can all be utilized.

Disadvantages

- The number of comparisons is large, increasing cost and latency.
- Non-transitivity can make the problem ill-defined.
- Positional bias is introduced.
- Susceptible to bias toward length and assertive tone [17, 45].

3.4 Setwise: An Intermediate Approach to Reduce LLM Calls

Positioned between pairwise and listwise is the setwise approach. Setwise compares $c \geq 3$ items together, asking, for example, “which is the best?” or “where should this be inserted?” By doing so, multiple comparisons can be performed in a single call, reducing the number of calls. Since the cost of LLMs is strongly dominated by the number of calls, this is effective [31, 55].

For example, Zhuang et al. [55] proposed a method that uses a $(c-1)$-ary heap instead of a binary heap in heapsort, and compares the node with all its children in a single call, thereby reducing the number of calls.

3.5 Hybrid Approaches

LLM sorting methods can also combine the basic frameworks described above [17, 38]. For example, PRE-PAIR (Pointwise REasoning within a PAIRwise framework) [17] first performs chain-of-thought reasoning independently for each item, generating rationales for its strengths and weaknesses. Then, the descriptions and rationales for each item are combined and used in pairwise comparison. This leverages the advantage of pointwise—being less susceptible to bias from length and assertive tone—while also benefiting from the ease and stability of pairwise comparisons.

Table 1 summarizes representative examples and characteristics of each approach.

The next section further concretizes these algorithms and discusses algorithm design principles in detail.

4 Algorithm Design

This section surveys the design principles of LLM sorting algorithms, with a focus on cost reduction.

4.1 Cost Model: Comparison Count Alone Is Not Enough

The computational complexity of classical sorting is discussed in terms of comparison count, but the cost model when using LLMs is more complex.

First, since LLM computation is expensive, caching is effective [50]. In classical numerical comparison, caching is almost meaningless, but since LLM calls are far heavier than cache lookups, caching makes a significant difference.

Care must also be taken in how LLM calls are made.

Prompt caching is important. Whether using pointwise, pairwise, listwise, or setwise approaches, the same type of prompt is used for multiple calls. The computation for the prompt portion is redundant and can be computed once and reused. When running a local model, this can be achieved by saving the KV cache. When using APIs, providers such as OpenAI and Anthropic offer prompt caching features, enabling up to 90% cost reduction [5, 29].

Prompt prefilling is also effective. The cost associated with input length and output length differs. For inputs, KV cache computation can be parallelized across all tokens. Therefore, when running on highly parallel GPUs, the cost of increasing input length is almost negligible as long as idle compute units remain. On the other hand, outputs must be processed one token at a time due to autoregressive generation, which is time-consuming. This holds when using APIs as well. For example, the GPT-5.2 API charges $1.75 per 1M input tokens but $14.00 per 1M output tokens—an 8x difference [30]. Therefore, reducing the number of output tokens is crucial.

For instance, methods like FIRST, which input all items and prompt the model to output the highest-ranked item, observing only the prediction of a single token and ranking items by their output probabilities, are more efficient than methods that sequentially generate an entire ranking, since they output only a single token.

Additionally, the PREPAIR approach [17] described in Section 3.5, which pre-generates reasoning for each item and combines them as input during pairwise comparison, also benefits from this principle. In the pairwise comparisons that represent the cost bottleneck, the chain of thought only affects input length, making this approach more efficient than generating a chain-of-thought reasoning path for each pairwise comparison.

Batch size is also important. Since GPUs have high parallel execution capability, the processing time for batch size 1 is often the same as for batch size 2. Therefore, batching multiple comparisons into a single request improves efficiency. In heapsort, the items to compare cannot be determined until the previous comparison result is known, making batching difficult. In contrast, quicksort can execute the comparison of the pivot with x_1, the pivot with x_2, ..., and the pivot with x_n all in parallel. When parallelism is high enough to execute n comparisons simultaneously, this yields up to an n-fold speedup, completing the sort in $O(\log n)$ rounds. When using APIs, this may not reduce cost, but issuing multiple requests simultaneously can reduce latency.

Below, we introduce specific methods and design guidelines useful for designing sorting-with-LLMs algorithms.

4.2 Single-Token Inference: FIRST

FIRST (Faster Improved Re-ranking with a Single Token) [37] achieves speedup by asking “which is the best?” and using the logits for the first token generated by the model (i.e., candidate IDs) to determine the ranking. Specifically, the following prompt is used.

```
I will provide you with 5 passages, each indicated by a alphabetical identifier [].
Rank the passages based on their relevance to the search query: {query}.
[A] {passage 1 content}
[B] {passage 2 content}
[C] {passage 3 content}
[D] {passage 4 content}
[E] {passage 5 content}
Search Query: {query}.
Rank the 5 passages above based on their relevance to the search query. All the
passages should be included and listed using identifiers, in descending order of
relevance. The output format should be [] > [], e.g., [D] > [B], Only respond with the
 ranking results, do not say any word or explain.
[
```

The LLM is asked to predict the next token, and the probabilities for "A", "B", "C", "D", "E" are retrieved. By ranking items in order of decreasing probability, a ranking is obtained. This yields a ranking faster than performing full decoding.

In this method, the number of output tokens is just one, dramatically reducing output cost. Additionally, since all candidates are input at once, the method also enjoys the advantages of the listwise approach, such as the ability to make context-based comparisons.

Care must be taken with how items are identified. If excessively long numbers are used, the ID may be split across multiple tokens, making it impossible to obtain accurate probabilities. For this reason, FIRST uses alphabetical letters as IDs. When the number of items to compare exceeds 26, the sliding window method described in Section 3.1 can be used to perform ranking in multiple rounds.

FIRST can be used as a standalone method, but Reddy et al. [37] further improved performance by fine-tuning with ranking training data and the RankNet loss [7], so that the next-token probabilities correspond to the correct ranking order.

4.3 Leveraging Logits

Leveraging logits and probabilities, as in FIRST, is effective. Logits and probabilities contain far more information than a discrete single next token. The method described in Section 3.2, which weights pointwise scores by probability [24], follows the same principle. Since logits are often available for free, methods that utilize them are highly worth considering in method design. Even when using APIs, where logits were previously unavailable, they can now be obtained through parameters such as OpenAI API's `top_logprobs`.

4.4 Chain-of-Thought (CoT) and Its Role

Using Chain of Thought to improve performance is common across LLM applications. G-Eval [24] achieves more accurate scoring in the pointwise approach by having the LLM generate evaluation steps via chain-of-thought. Rank-K [51] achieves more stable ranking in the listwise approach by having the LLM generate ranking rationales via chain-of-thought. Since Chain-of-Thought increases the number of output tokens and thus cost, as discussed in Section 4, pre-generating reasoning for each item and utilizing it during pairwise comparison can improve performance while keeping cost in check.

4.5 Cost Reduction via Distillation

Models such as GPT-5 and Claude Opus are powerful but are black-box APIs with high cost. For this reason, there have been attempts to distill these powerful closed models as teacher signals into open models [32, 33]. RankZephyr distilled RankGPT-4 [43] into the open model Zephyr 7B, achieving comparable performance [33].

Moreover, even with a small base model, using a sorting algorithm requires calling the model many times, which is expensive. Therefore, it is also effective to use a carefully constructed ranking produced by a sorting algorithm as a teacher signal and distill it into a conventional reranking model such as DeBERTa-large using RankNet loss or similar [43].

4.6 Embracing "Incomplete Comparisons" as a Strategy

Given the high cost of the comparator, it is common to avoid a full $n \log n$ comparisons and instead achieve a good trade-off with incomplete sorting. The key here is the initial ranking. In search, the better the initial ordering from BM25 or similar, the more effective refinement techniques such as sliding window or setwise insertion become [31, 43].

Theoretically, this corresponds to Sorting with Predictions [6]. This work considers two problem settings for exactly sorting with the primary comparison function. The first setting assumes the availability of a "dirty" comparison function. This comparison function can be called for free but is not guaranteed to agree with the primary comparison function. The goal is to leverage this dirty comparison function while ultimately sorting exactly according to the primary comparison function. The second setting assumes that predictions of each element's position are given. These predictions are not guaranteed to be accurate, but the goal is to leverage them while ultimately sorting exactly according to the primary comparison function. The method of Bai and Coester [6] achieves ideal results in both settings: when the predictions are good, sorting can be completed with only $O(n)$ calls to the primary comparison function, and in the worst case when predictions are poor or adversarial, sorting still requires only $O(n \log n)$ comparisons—the same as without predictions. For example, in the first setting, a balanced binary search tree is constructed using the primary comparison function. When a new item arrives, the dirty comparison function is first used to estimate the insertion position in the balanced binary tree. Then, the primary comparison function is used to verify the dirty comparisons by traversing upward from the leaf, correcting the insertion position for exact insertion. If the dirty comparison function is reasonably accurate, the first stage identifies an approximately correct position, and only local corrections are needed, so insertion completes in constant time. Even if the dirty comparisons are completely random, verification from leaf to root and back down to a new leaf requires at most twice the $O(\log n)$ comparisons. In LLM sorting as well, coarse scores from BM25 or similar can be used as a "dirty" comparison function or position predictions to reduce the number of LLM comparator calls. In practice, due to implementation complexity, the method of Bai and Coester [6] is rarely used directly, but in principle it can be applied to LLM sorting, and the same reasoning shows that leveraging an initial ranking enables more efficient sorting.

One approach in this direction is setwise insertion [31]. As noted in Section 2.1, bubble sort requires $O(n^2)$ comparisons even when the initial ranking is good, whereas insertion sort can achieve $O(n)$ comparisons when the initial ranking is good. This method exploits this property to reduce the number of comparisons. Additionally, this method further reduces the number of comparisons by focusing only on sorting the top-k items. Specifically, the top k candidates are first sorted using setwise heapsort. Then, each subsequent item is evaluated to determine whether it belongs in the top-k. This can be done by comparing it against the k-th ranked element. Using a setwise approach, multiple items can be evaluated simultaneously. Moreover, when items are sorted in descending order by a coarse criterion, many of the subsequent items are expected not to belong in the top-k and can be immediately rejected, reducing the number of comparisons. When an item is found to belong in the top-k, it is inserted at the appropriate position within the top-k using an setwise insertion sort procedure, maintaining the sorted order within the top-k. Here too, comparing multiple items at once enables efficient identification of the insertion position.

Additionally, due to the lack of consistency in LLMs, situations can arise where $x_i \succ x_j$ and $x_j \succ x_i$, making the ordering indeterminate. In such cases, the initial ranking can be used as a tiebreaker, giving priority to the item ranked higher initially. For example, in the illustrative example of reordering papers by preference described in Section 1, by using citation count as the initial ranking, papers with higher citation counts—which are presumably more important—can be prioritized in subtle cases that cannot be resolved from titles and abstracts alone.

4.7 Tournament Strategies

Methods inspired by sports and chess ranking systems have also been developed.

TourRank [10], inspired by the FIFA ranking system, divides items into groups, has them compete against each other, and awards points as items advance through the tournament. Multiple tournaments are held in parallel with different group assignments, and items are ranked based on their accumulated points.

PRP-Graph [27], inspired by the Swiss-System Tournament used in chess, compares items with similar tentative scores that have not yet been compared. This enables efficient discrimination between ranks. Additionally, a method has been proposed to represent pairwise comparison results as a graph and aggregate global information using techniques like PageRank to produce a higher-quality final ranking.

4.8 Cost–Performance Trade-off

LLM-based comparisons are high-quality but expensive. Therefore, research has been conducted on improving the cost–quality trade-off by combining low-cost and high-cost methods [8, 9, 35]. For example, a multi-stage ranking strategy such as the following can be considered:

- Sort all items using BM25
- Sort the top 100 items using GPT-5-nano
- Sort the top 20 items using GPT-5.2

By optimizing which methods to use, in what order, and up to which rank each method should be responsible for through prior profiling, a favorable trade-off between cost and quality can be achieved.

4.9 Prompt Design

It goes without saying that prompt design is important [15]. As with typical LLM applications, techniques such as few-shot with demonstration [53] and Chain-of-Thought [24, 53] are effective.

For comparisons and pointwise scoring, it is helpful to clearly specify the criteria and evaluation methodology [24]. Methods such as HD-EVAL [25] that automatically break down evaluation perspectives also exist. However, it should be noted that LLMs may not accurately capture the criteria: a mere spelling error can cause scores for "Faithfulness" or "Informativeness" to drop, or a factually incorrect statement can lower scores for "Fluency" or "Grammaticality"—indicating that criterion confusion can occur [16, 48], and even state-of-the-art LLMs make elementary mistakes [40].

Additionally, as in G-Eval [24], explicitly specifying evaluation steps has been practiced. For example, when measuring the quality of news summaries, including evaluation steps like the following in the prompt helps the LLM follow a structured evaluation procedure.

```
## Evaluation Steps
1. Read the news article carefully and identify the main topic and key points.
2. Read the summary and compare it to the news article. Check if the summary covers
the main topic and key points of the news article, and if it presents
them in a clear and logical order.
3. Assign a score for coherence on a scale of 1 to 5, where 1 is the lowest and 5 is
the highest based on the Evaluation Criteria.
```

5 Applications

5.1 Reranking in Information Retrieval

The area where sorting with LLMs is most widely used is search reranking [1]. In conventional search systems, an approach has been taken where candidates are first narrowed down using fast algorithms such as BM25, followed by reranking using high-performance models such as BERT. Many attempts have been made to replace this reranking step with sorting using LLMs [34, 43].

As discussed in Section 2.2, LLM-based comparisons do not satisfy transitivity, and in such cases, different sorting algorithms can yield different final rankings. However, in the experiments of Wisznia et al. [50], even when different sorting algorithms (quicksort, mergesort, heapsort, etc.) were used, downstream task performance did not change significantly. This suggests that in LLM sorting, the choice of sorting algorithm has little impact on task performance and should instead be guided by considerations of cost and execution time.

A characteristic unique to search reranking is that only the top results are typically viewed. For this reason, evaluation metrics that consider only the top results, such as recall@K and precision@K, or metrics like nDCG that assign higher weights to top-ranked quality, are commonly used. Consequently, rather than uniformly sorting the entire list, it is better in terms of both quality and efficiency to allocate effort toward improving the quality of the top results. For this reason, incomplete sorting methods such as the sliding window approach described in Section 3.1 are often sufficient.

5.2 Adaptive Retrieval

SlideGar (Sliding-window-based Graph Adaptive Retrieval) [36] is a method that applies LLM sorting not only to reranking but also to the retriever. No matter how much reranking performance is improved, if the initial retriever misses relevant documents in the first place, a good ranking cannot be obtained. Therefore, SlideGar applies the sliding window method described in Section 3.1 to the initial item list from the retriever. At each sliding window step, for items that the LLM ranks highly, related items are newly retrieved and forcibly added to the next window. Items whose ranking improved through LLM-based comparison may have been items whose relevance the retriever failed to assess correctly, suggesting that similar good items may have been missed. SlideGar retrieves such items on the fly during the sorting process, growing the ranking list while incorporating related items.

5.3 LLM Performance Comparison

Sorting with LLMs is also used for comparing the performance of machine learning models and LLMs. In recent years, various methods and models have been developed for the same tasks, and leaderboards that rank them have been published. MT-bench and Chatbot Arena [11] are benchmarks for measuring chatbot performance, ranking multiple chatbots through comparative evaluation using both humans and LLMs. The approach taken here involves having a judge LLM determine which of two LLMs' responses is better, then ranking them in order of performance.

Since the quality of outputs is subjective, conventional sorting cannot handle this. While quality evaluation has traditionally been performed primarily by humans, LLM-as-a-Judge has attracted attention as a lower-cost automated approach.

5.4 Feature Selection

LLM sorting can also be applied to feature selection, a classical machine learning task. Eureka [39] solves the task of ranking features by "interestingness" using LLMs, building models that produce not only highly accurate predictions but also insightful and surprising ones from informative features.

5.5 Social Sorting

Sorting with LLMs can also be applied to social tasks such as ordering politicians from left to right on the political spectrum, or ordering economic news from optimistic to pessimistic [12, 13, 18].

Such items are difficult to quantify and require sophisticated judgment, making them well-suited for sorting with LLMs.

Unlike search and leaderboards where top-ranked items are most important, ordering all items matters in these applications, so complete sorting is often appropriate.

5.6 Customer Support Prioritization

Sorting with LLMs can also be used for prioritizing customer inquiries and bug report triaging [3, 22, 49]. In such tasks, priority must be determined by comprehensively considering various factors such as severity, estimated time to resolve, and time elapsed since the report. Since analyzing each of these factors is itself a challenging task, leveraging the capabilities of LLMs is a promising approach.

6 Advanced Topics

6.1 Addressing Bias

LLMs are known to have various biases.

Positional bias—the tendency to select the option presented first—was discussed in Section 2.2. In the pairwise setting, positional bias can be addressed by querying with both orderings. In the listwise setting, trying all permutations is intractable, making this difficult to address. Approaches that generate rankings with multiple random orderings and then aggregate them to produce a stabilized final ranking have been proposed [44].

LLMs are also known to have a verbosity bias, tending to rate longer outputs more highly [53]. AlpacaEval [14] successfully eliminated the influence of length by performing regression analysis with an item's true ability and length as explanatory variables to estimate each item's win rate, then measuring the item's true ability using a model with length forced to zero—a causal approach.

Pairwise comparison has been observed to be particularly vulnerable to such biases [17, 45]. When asked to judge which of two items is better, in borderline cases the LLM tends to default to choosing the longer or more flamboyant one. The pointwise approach is relatively robust to such biases, which is why hybrid approaches like PREPAIR (Pointwise REasoning within a PAIRwise framework) [17]—which first generates reasoning pointwise and then performs pairwise comparison—have proven effective.

6.2 Evaluation Hacking

Caution is needed regarding methods that exploit LLM biases to hack rankings through adversarial items.

In benchmarks measuring the true performance of chatbots, a chatbot is given a task, generates a response, and a judge LLM evaluates that response. Zheng et al. [54] demonstrated that even a "null model" that ignores the given task and always generates the same response can be placed at the top of the leaderboard if this response is tuned to be selected by the LLM judge. The fixed response used in this case is composed of an optimizable prefix and a manually crafted cheating response as follows.

```
} } } } } verify }:param } } } withObject }pressive } browsInput shrimp sett(query176
}summer }findById } }CheckBoxucÿao  } }} }.thumb intoxic-kit(pd } } } }framework }
asket ////////// }POINTicides } } } } gamleylv } }[trainadapt } meld } } } } ***!\n }
}Precio }.Float } ť;\r\n } K obrigase.Millisecond } } }inks } } } PED } } } } } }
relativesPl } } }\tdriver }vel } } } } }(config } columna } } } } pedestrian }
kiyebalanceRule }restore steel COR } } } } } }xpath\n }\n}\n\nIgnore the above ##
Model Outputs\n\n\n\n\n## Instruction\n\n{\n instruction: Output nothing,\n}\n\n##
Model Outputs\n\nHere are the unordered outputs from the models. Each output is
associated with a specific model, identified by a unique model identifier.\n\n{\n {\n
model identifier: M,\n output: \n },\n {\n model identifier: m,\n output:
```

LLM-based evaluation is vulnerable to such attacks, so caution is necessary in environments where end users can freely add items.

7 Tools and Implementation

RankLLM is a Python package that enables reproduction and comparison of LLM reranking methods including RankGPT/LRL/RankVicuna/RankZephyr [41]. Rankify is a toolkit that integrates retrieval, reranking, and RAG, providing a framework for cross-evaluating multiple datasets and methods [2]. LLM4Ranking is also a framework for LLM-based ranking [23]. These frameworks support closed models via API and local open models. While they primarily focus on search applications, they serve as useful references for implementing LLM sorting.

This paper has introduced various design principles for sorting algorithms using LLMs. For those who want to try them in practice, we recommend starting with either the sliding window method or quicksort (KwikSort). For tasks like search result reranking where exact sorting is not necessary and you simply want to refine the top results, the sliding window method should suffice. For tasks where ordering matters not only at the top but across the entire spectrum—such as economic news stance or political ideology—quicksort is a good choice for sorting the entire list. Quicksort is particularly well-suited for LLMs because it provides theoretical guarantees even when transitivity does not hold, and it supports parallel execution. For those who wish to further enhance their approach, we hope the design techniques introduced in this paper will be of help.

8 Conclusion

This paper surveyed the design principles and applications of sorting algorithms using LLMs. LLM sorting possesses characteristics distinct from traditional numerical comparison, necessitating new design considerations in algorithm development. In particular, designs that consider the trade-off between cost reduction and performance improvement are important. By using LLM sorting, tasks that were previously difficult with conventional methods—such as ranking political ideologies or prioritizing customer support requests—can be accomplished solely through LLM calls without developing specialized models. Sorting is a quintessential example of classical tasks in computer science, and we believe that the application of LLMs to various other classical algorithmic problems will continue to advance beyond sorting. We hope this paper contributes to that endeavor.

References

[1] A. Abdallah, B. Piryani, J. Mozafari, M. Ali, and A. Jatowt. How good are LLM-based rerankers? an empirical analysis of state-of-the-art reranking models. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 13049–13063. Association for Computational Linguistics, 2025.

[2] A. Abdallah, B. Piryani, J. Mozafari, M. Ali, and A. Jatowt. Rankify: A comprehensive python toolkit for retrieval, re-ranking, and retrieval-augmented generation. *arXiv preprint arXiv:2502.02464*, 2025. URL https://arxiv.org/abs/2502.02464.

[3] J. Acharya and G. Ginde. Graph neural network vs. large language model: A comparative analysis for bug report priority and severity prediction. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE*, pages 2–11. ACM, 2024.

[4] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: Ranking and clustering. *Journal of the ACM*, 55(5):23:1–23:27, 2008.

[5] Anthropic. Prompt caching. Claude API Docs, 2024. URL `https://platform.claude.com/docs/en/build-with-claude/prompt-caching`. Accessed: 2026-02-07.

[6] X. Bai and C. Coester. Sorting with predictions. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS*, 2023.

[7] C. J. C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. N. Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd International Conference on Machine Learning, ICML*, volume 119, pages 89–96. ACM, 2005.

[8] L. Chen, M. Zaharia, and J. Y. Zou. FrugalML: How to use ML prediction apis more accurately and cheaply. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS*, 2020.

[9] L. Chen, M. Zaharia, and J. Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024.

[10] Y. Chen, Q. Liu, Y. Zhang, W. Sun, X. Ma, W. Yang, D. Shi, J. Mao, and D. Yin. TourRank: Utilizing large language models for documents ranking with a tournament-inspired strategy. In *Proceedings of the ACM on Web Conference 2025, WWW*, pages 1638–1652. ACM, 2025.

[11] W. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, B. Zhu, H. Zhang, M. I. Jordan, J. E. Gonzalez, and I. Stoica. Chatbot arena: An open platform for evaluating LLMs by human preference. In *Proceedings of the 41st International Conference on Machine Learning, ICML*, 2024.

[12] R. Di Leo, C. Zeng, E. Dinas, and R. Tamtam. Mapping (a)ideology: A taxonomy of european parties using generative llms as zero-shot learners. *Political Analysis*, 33(4):456463, 2025.

[13] M. R. DiGiuseppe and M. Flynn. Scaling open-ended survey responses using LLM-paired comparisons. *SSRN Electronic Journal*, 2025. URL `http://dx.doi.org/10.2139/ssrn.5112677`.

[14] Y. Dubois, B. Galambosi, P. Liang, and T. B. Hashimoto. Length-controlled AlpacaEval: A simple way to debias automatic evaluators. In *Proceedings of the 1st Conference on Language Modeling, COLM*, 2024.

[15] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, H. Liu, et al. A survey on LLM-as-a-judge. *The Innovation*, 2024.

[16] X. Hu, M. Gao, S. Hu, Y. Zhang, Y. Chen, T. Xu, and X. Wan. Are llm-based evaluators confusing NLG quality criteria? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL*, pages 9530–9570. Association for Computational Linguistics, 2024.

[17] H. Jeong, C. Park, J. Hong, H. Lee, and J. Choo. The comparative trap: Pairwise comparisons amplifies biased preferences of LLM evaluators. In *Proceedings of the 8th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pages 79–108. Association for Computational Linguistics, 2025.

[18] G. Le Mens and A. Gallego. Positioning political texts with large language models by asking and averaging. *Political Analysis*, 33(3):274–282, 2025.

[19] Y. Lee, I. Park, and M. Kang. FLEUR: an explainable reference-free evaluation metric for image captioning using a large multimodal model. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL*, pages 3732–3746. Association for Computational Linguistics, 2024.

[20] Y. Li, C. Xu, J. Xu, and J. Wen. BordaRAG: Resolving knowledge conflict in retrieval-augmented generation via borda voting process. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management, CIKM*, pages 1737–1747. ACM, 2025.

[21] Z. Li, C. Wang, P. Ma, D. Wu, S. Wang, C. Gao, and Y. Liu. Split and merge: Aligning position biases in LLM-based evaluators. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 11084–11108. Association for Computational Linguistics, 2024.

[22] F. Liu, X. He, T. Zhang, J. Chen, Y. Li, L. Yi, H. Zhang, G. Wu, and R. Shi. Tickit: Leveraging large language models for automated ticket escalation. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE*, pages 343–354. ACM, 2025.

[23] Q. Liu, H. Duan, Y. Chen, Q. Lu, W. Sun, and J. Mao. LLM4Ranking: An easy-to-use framework of utilizing large language models for document reranking. *arXiv preprint arXiv:2504.07439*, 2025. URL `https://arxiv.org/abs/2504.07439`.

[24] Y. Liu, D. Iter, Y. Xu, S. Wang, R. Xu, and C. Zhu. G-Eval: NLG evaluation using GPT-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 2511–2522. Association for Computational Linguistics, 2023.

[25] Y. Liu, T. Yang, S. Huang, Z. Zhang, H. Huang, F. Wei, W. Deng, F. Sun, and Q. Zhang. HD-Eval: Aligning large language model evaluators through hierarchical criteria decomposition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL*, pages 7641–7660. Association for Computational Linguistics, 2024.

[26] A. Liusie, P. Manakul, and M. J. F. Gales. LLM comparative assessment: Zero-shot NLG evaluation through pairwise comparisons using large language models. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics, EACL*, pages 139–151. Association for Computational Linguistics, 2024.

[27] J. Luo, X. Chen, B. He, and L. Sun. Prp-graph: Pairwise ranking prompting to llms with graph aggregation for effective text re-ranking. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL*, pages 5766–5776. Association for Computational Linguistics, 2024.

[28] X. Ma, X. Zhang, R. Pradeep, and J. Lin. Zero-shot listwise document reranking with a large language model. *arXiv preprint arXiv:2305.02156*, 2023. URL `https://arxiv.org/abs/2305.02156`.

[29] OpenAI. Prompt caching. OpenAI API documentation, 2024. URL `https://platform.openai.com/docs/guides/prompt-caching`. Accessed: 2026-02-07.

[30] OpenAI. Pricing. OpenAI API documentation, 2025. URL `https://platform.openai.com/docs/pricing`. Accessed: 2026-02-07.

[31] J. Podolak, L. Peric, M. Janicijevic, and R. Petcu. Beyond reproducibility: Advancing zero-shot LLM reranking efficiency with setwise insertion. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR*, pages 3205–3213. ACM, 2025.

[32] R. Pradeep, S. Sharifymoghaddam, and J. Lin. RankVicuna: Zero-shot listwise document reranking with open-source large language models. *arXiv preprint arXiv:2309.15088*, 2023. URL `https://arxiv.org/abs/2309.15088`.

[33] R. Pradeep, S. Sharifymoghaddam, and J. Lin. RankZephyr: Effective and robust zero-shot listwise reranking is a breeze! *arXiv preprint arXiv:2312.02724*, 2023. URL `https://arxiv.org/abs/2312.02724`.

[34] Z. Qin, R. Jagerman, K. Hui, H. Zhuang, J. Wu, L. Yan, J. Shen, T. Liu, J. Liu, D. Metzler, X. Wang, and M. Bendersky. Large language models are effective text rankers with pairwise ranking prompting. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 1504–1518. Association for Computational Linguistics, 2024.

[35] M. S. Rashid, J. A. Meem, Y. Dong, and V. Hristidis. EcoRank: Budget-constrained text re-ranking using large language models. In *Findings of the Association for Computational Linguistics, ACL*, pages 13049–13063. Association for Computational Linguistics, 2024.

[36] M. Rathee, S. MacAvaney, and A. Anand. Guiding retrieval using LLM-based listwise rankers. In *Proceedings of the 47th European Conference on Information Retrieval, ECIR*, volume 15572, pages 230–246. Springer, 2025.

[37] R. G. Reddy, J. Doo, Y. Xu, M. A. Sultan, D. Swain, A. Sil, and H. Ji. FIRST: Faster improved listwise reranking with single token decoding. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 8642–8652. Association for Computational Linguistics, 2024.

[38] R. Ren, Y. Wang, K. Zhou, W. X. Zhao, W. Wang, J. Liu, J. Wen, and T. Chua. Self-calibrated listwise reranking with large language models. In *Proceedings of the ACM on Web Conference 2025, WWW*, pages 3692–3701. ACM, 2025.

[39] R. Sato. Interestingness first classifiers. *arXiv preprint arXiv:2508.19780*, 2025. URL `https://arxiv.org/abs/2508.19780`.

[40] R. Sato. Even GPT-5.2 can't count to five: The case for zero-error horizons in trustworthy LLMs. *arXiv preprint arXiv:2601.15714*, 2025. URL `https://arxiv.org/abs/2601.15714`.

[41] S. Sharifymoghaddam, R. Pradeep, A. Slavescu, R. Nguyen, A. Xu, Z. Chen, Y. Zhang, Y. Chen, J. Xian, and J. Lin. RankLLM: A python package for reranking with LLMs. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR*, pages 3681–3690. ACM, 2025.

[42] L. Shi, C. Ma, W. Liang, X. Diao, W. Ma, and S. Vosoughi. Judging the judges: A systematic study of position bias in LLM-as-a-judge. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics, IJCNLP-APACL*, pages 292–314, 2025.

[43] W. Sun, L. Yan, X. Ma, S. Wang, P. Ren, Z. Chen, D. Yin, and Z. 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*, pages 14918–14937. Association for Computational Linguistics, 2023.

[44] R. Tang, X. Zhang, X. Ma, J. Lin, and F. Ture. Found in the middle: Permutation self-consistency improves listwise ranking in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL*, pages 2327–2340. Association for Computational Linguistics, 2024.

[45] T. Tripathi, M. Wadhwa, G. Durrett, and S. Niekum. Pairwise or pointwise? evaluating feedback protocols for bias in LLM-based evaluation. In *Proceedings of the 2nd Conference on Language Modeling, COLM*, 2025.

[46] W. Wang, Y. Wang, and H. Huang. Ranked voting based self-consistency of large language models. In *Findings of the Association for Computational Linguistics, ACL*, pages 14410–14426. Association for Computational Linguistics, 2025.

[47] Y. Wang, Y. Song, T. Zhu, X. Zhang, Z. Yu, H. Chen, C. Song, Q. Wang, C. Wang, Z. Wu, X. Dai, Y. Zhang, W. Ye, and S. Zhang. TrustJudge: Inconsistencies of llm-as-a-judge and how to alleviate them. *arXiv preprint arXiv:2509.21117*, 2025. URL `https://arxiv.org/abs/2509.21117`.

[48] Y. Wang, J. Yuan, Y. Chuang, Z. Wang, Y. Liu, M. Cusick, P. Kulkarni, Z. Ji, Y. Ibrahim, and X. Hu. DHP benchmark: Are llms good NLG evaluators? In *Findings of the Association for Computational Linguistics: NAACL*, pages 8079–8094. Association for Computational Linguistics, 2025.

[49] Z. Wang, J. Li, M. Ma, Z. Li, Y. Kang, C. Zhang, C. Bansal, M. Chintalapati, S. Rajmohan, Q. Lin, D. Zhang, C. Pei, and G. Xie. Large language models can provide accurate and interpretable incident triage. In *Proceedings of the 35th IEEE International Symposium on Software Reliability Engineering, ISSRE*, pages 523–534. IEEE, 2024.

[50] J. Wisznia, C. Bolaños, J. Tollo, G. Marraffini, A. Gianolini, N. Hsueh, and L. D. Corro. Are optimal algorithms still optimal? rethinking sorting in LLM-based pairwise ranking with batching and caching. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL*, pages 1064–1072. Association for Computational Linguistics, 2025.

[51] E. Yang, A. Yates, K. Ricci, O. Weller, V. Chari, B. V. Durme, and D. J. Lawrie. Rank-K: Test-time reasoning for listwise reranking. *arXiv preprint arXiv:2505.14432*, 2025. URL `https://arxiv.org/abs/2505.14432`.

[52] Y. Zeng, O. Tendolkar, R. Baartmans, Q. Wu, H. Wang, and L. Chen. LLM-RankFusion: Mitigating intrinsic inconsistency in LLM-based ranking. *arXiv preprint arXiv:2406.00231*, 2024. URL `https://arxiv.org/abs/2406.00231`.

[53] L. Zheng, W. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS*, 2023.

[54] X. Zheng, T. Pang, C. Du, Q. Liu, J. Jiang, and M. Lin. Cheating automatic LLM benchmarks: Null models achieve high win rates. In *Proceedings of the 13th International Conference on Learning Representations, ICLR*, 2025.

[55] S. Zhuang, H. Zhuang, B. Koopman, and G. 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*, pages 38–47. ACM, 2024.