Ensemble Learning for Large Language Models in Text and Code Generation: A Survey

***Abstract*—Generative pretrained transformers (GPTs) are the common large language models (LLMs) used for generating text from natural language inputs. However, the fixed properties of language parameters in individual LLMs can lead to inconsistencies in the generated outputs. This limitation also restricts the models' ability to represent diverse language patterns due to inherent biases. Moreover, many powerful LLMs are closed-source. This prevents organizations from integrating their data into these systems, raising concerns about data privacy and limiting industry applications. Inspired by the successful application of LLM ensemble models in text generation, recent literature has also investigated their potential in code generation. This article reviews these emerging LLM ensemble approaches. Our goal is to enhance readers' understanding of existing techniques and encourage further research and practical implementation, aiming to expand the real-world applications of LLM ensemble models in both text and code generation. We categorize these approaches into seven main methods: weight merging, knowledge fusion, mixture of experts, reward ensemble, output ensemble, routing, and cascading. From this list, we focus on four methods and models that show strong performance and potential for broader applications. We analyze their modeling steps, training methods, and output features to provide a clear understanding of their capabilities. Our findings highlight the benefits of LLM ensemble techniques. These include better representation of diversity, improved output quality, and greater flexibility in applications. This information offers valuable insights for selecting models for various real-world tasks involving text and code generation, and potentially applying methods to multimodal LLMs.**

***Impact Statement*—Generative pretrained transformers (GPTs) are widely used large language models (LLMs) for text and code generation, providing researchers with an adaptive fine-tuning framework that alleviates the burden of model training. However, recent studies indicate that the best LLMs achieve only 57% in chat generation quality due to their single architectural limitations and inability to capture diverse language patterns. This survey reviews LLM ensemble methods that address these shortcomings, resulting in a notable increase in instruction-following accuracy to 65%. These ensemble techniques enhance diversity, output quality, and flexibility, making them suitable for various applications such as reasoning, question-answering, and code generation. The insights gained from this research are crucial for selecting effective models for real-world tasks. Additionally, LLM ensemble models can transform decision-making processes across sectors like medicine, finance, education, and customer service, enabling businesses to optimize operations, improve user experiences, and reduce costs associated with traditional models.**

***Index Terms*—Large language models (LLMs), ensemble learning, natural language generation, code generation, transformer, generative pretrained transformers (GPTs).**

I. Introduction

MANY applications in language processing have been framed as generative language modeling tasks. One common task is instruction following, which predicts conversational word sequences from language feature sequences. Many generative tasks in natural and programming language processing aim to generate language features from various types of inputs. They are referred to as text and code generation tasks in this article. These tasks cover a wide range of research areas, including mathematical reasoning, question answering (QA), massive multitask language understanding (MMLU), and code generation. These topics have the common goal of generating language features and differ in the forms of inputs.

The popular LLM for text and code generation is the single GPT [1], [2]. In the context of language generation, the GPT is a generative pretrained transformer model that can represent language features using a sequence of text and position embeddings linked to linguistic characteristics. Language generation approaches using GPT have demonstrated the ability to produce highly intelligible and natural text. [2]–[4]. However, the generated responses often exhibit noticeable biases [5] compared to majority-voted answers. Lack of consistency of language models in their responses due to biased training data is one of the main reasons for this sensitivity [6].

For instance, in GPT-based instruction following, fine-tuned, context-dependent GPTs are used to represent language feature embeddings based on conversational texts [2]. The embedding transitions through each layer of the transformer are characterized by attention weights. During training, the parameters of the GPT are estimated based on the cross entropy (CE) and human feedback criterion. At synthesis time, given an input sentence and the trained parameters, the most likely language features are predicted using a meta-generation algorithm [7], [8]. Although this process enhances the robustness of parameter estimation and generation, the single-LLM approach has notable limitations.

Firstly, the fixed properties of language parameters in individual LLMs can lead to inconsistencies in output [9] and limit the model's ability to express diverse language patterns due to inherent biases [10]. We identify this issue stems from two efficiency assumptions: the efficiency of information retrieval within a fixed context window and the efficiency of implicit knowledge representation without direct access to external knowledge bases. Consequently, the generated language features often depend heavily on the LLM, leading to hallucinated information and degraded text quality [11]. In recent years, various techniques have been proposed to address these model-dependent issues. These include introducing improved language models (e.g., the mixture of experts LSTM [12], herd of LLMs [13], and multimodal LLM [14], [15]), enhancing training criteria (e.g., scalable knowledge distillation training [16]), and modifying the meta-generation algorithm (e.g., increasing computational capacity using intermediate decoding steps [17]). Secondly, many powerful LLMs have closed-

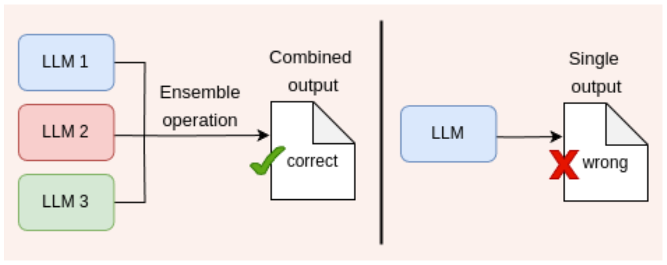

Fig. 1. The difference between an ensemble LLM and a single LLM. The single LLM can limit generalization ability.

source code, which prevents organizations from integrating their data into these systems and raises data privacy concerns, limiting industrial applications.

Since 2022, LLM ensemble learning has emerged as a significant area of machine learning (ML) research, attracting the attention of many language modeling researchers [18], [19]. LLM ensemble learning refers to a class of ML techniques that exploit multiple LLMs of combined information processing to enhance feature generation, pattern recognition and classification. Researchers have explored both architecture-level and model-level ensemble methods. Architecture-level methods include techniques like weight merging [19], knowledge fusion [20], mixture of experts (MoEs) [21], and reward ensembles [22]. Model-level methods encompass output ensemble [23], routing [24], and cascading [9].

Strictly speaking, a MoE is a specific type of transformer model that incorporates multiple expert layers and serves as a base model to be combined in various LLM ensemble models (e.g., output ensemble models). MoEs, as self-attention models, often outperform conventional single LLMs, such as LLaMA 2 70B, while using fewer parameters [21]. Also, LLM ensemble models do not strictly adhere to traditional ensemble learning methods like bagging or boosting, but they function similarly to bias reduction models (e.g., weight merging [19] as bagging [25], cascading [9] as boosting [26]). Given their similar traits to conventional methods, we define our taxonomies as LLM ensemble in this article.

One example of the successful applications of LLM ensembling is code generation. In this approach, an ensemble of LLMs is introduced to replace a single LLM for evaluating the prediction of code snippets [21], [24], [27], as in Figure 1. LLM ensemble techniques have also been applied to the text generation recently to deal with the limitations of the conventional approaches [9], [18]–[20], [22]–[24], [28]–[63]. Different with the LLM ensembling in code generation is the data and model evaluation, these emerging language modeling approaches for text generation adopted various model structures. Some of them focus on improving the functions of weight merging [18], [19], [28]–[34], knowledge fusing [20], [44], or reward modeling [22], [47]–[49] using source LLMs or improving the gating functions to expert layers using MoEs [6], [64]. While some others combine the LLM output responses directly [23], [52]–[56] or model the entire routing process from input prompt to LLMs [9], [24], [57]–[62], [65].

LLM ensemble techniques outperform single-LLM models, like traditional GPTs, in representing diversity among LLM agents and output features. These techniques effectively combine varied strengths and features to enhance target responses, particularly in tasks like question answering. They utilize architecture-level frameworks, such as MoEs, and model-level frameworks, like output ensemble and routing. Additionally, human developers often use hierarchical structures to optimize from multiple solutions in code production [66], highlighting the need for a review of various LLM ensemble models for code generation applications.

This survey defines and describes LLM ensemble techniques based on original categorizations and subjectively evaluates selected models for their generation performance. From this categorization, we select four methods and models demonstrating strong performance and potential for broader applications. We analyze these models for text and code generation across three aspects: input and target features, model structures and training, and comparisons between text and code generation. This review seeks to improve readers' understanding of current techniques and encourage further research and practical implementation. We aim to broaden the real-world applications of LLM ensemble models in text and code generation, overcoming the limitations with single-LLM approaches.

While a few surveys touch on related topics, such as the work by Lu et al. [67] on LLM collaboration methods and Yang et al. [68] on ensemble techniques of LLMs and multimodal LLMs across various machine learning fields, our focus is more concentrated. We concentrate specifically on LLM ensemble methods for text and code generation, providing in-depth insights for practical applications.

This article first reviews the conventional and popular transformer framework for language generation, including GPT-based discriminative and generation tasks. We then analyze the limitations of these approaches and introduce key models and techniques relevant to LLM ensemble learning, such as MoEs, output ensemble, and routing. Finally, we review emerging language generation approaches using LLM ensemble techniques, discussing their motivations and implementations, and highlighting remaining challenges and future directions in this field.

II. Conventional Text Generation Using an LLM

The transformer model, introduced in 2017 [69], marked a significant advancement in natural language generation (NLG). This model is versatile, supporting various generative tasks across time series analysis, vision, and image processing. In NLG, transformers model the relationship between text and its linguistic realizations, enhancing performance across a broad range of applications. For instance, BERT utilizes bidirectional encoder representations to better understand sentence context [70], T5 treats all NLP tasks as text-to-text problems, facilitating its application in translation, summarization, and question answering [71], and Transformer XL extends the ability to learn dependencies beyond fixed lengths, improving handling of longer sequences [72].

Among these, the context-dependent transformers, specifically fine-tuned for particular tasks, have shown the success in NLG [73]. The Generative Pre-trained Transformer (GPT) is a prominent example of such models, known as a large language model. GPT models in language tasks can generate or classify a sequence of text using attention mechanism layers followed by blocks with high-dimensional inner states. The attention mechanism computes a weighted representation of input sequences, focusing on relevant contexts and dependencies to enhance language understanding and generation. Each block contains a feed-forward neural network (FFN) that processes the output from the attention mechanism to learn complex representations. An architectural example of a GPT is illustrated in Figure 2. The architecture of a GPT involves text and position embeddings that represent observed features, which are inputted into the transformer, and the transitions between layers are characterized by attention weights.

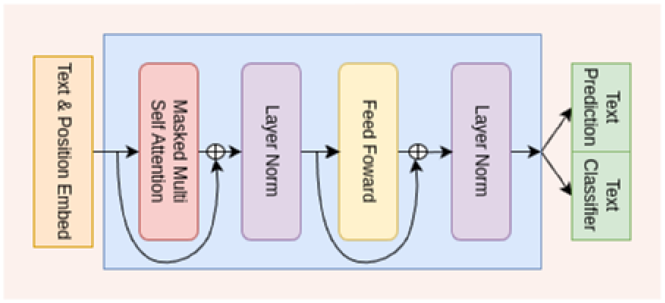

Fig. 2. Architecture of a GPT as a large language model, which is fine-tuned on a specific task given the text and position embeddings.

GPT-based language generation produces highly intelligible and natural texts. Compared to traditional recurrent-network-based methods, this attention-based approach is more efficient at handling long-range dependencies in language features, which correlate well to human perception.

Additionally, input transformation and fine-tuning techniques are used to train these models on various tasks, enhancing the diversity of their transfer performance [1], [11], [14], [74], [75]. The model pre-training is typically done using a next token prediction approach. At the training stage, a high-capacity language model with parameters Θ is first trained on a large corpus of text tokens $\mathcal{U} = \{u_1, ...u_n\}$ by maximizing the likelihood as

$$L_1(\mathcal{U}) = \sum_i \log\ P(u_i \mid u_{i-k}, ..., u_{i-1}; \Theta) \tag{1}$$

where P represents the probability of the current token given the previous tokens [73]. Given a sequence of input tokens, $x^1, ..., x^m$, the final transformer block's activation h is obtained. Next, a fine-tuning stage follows and adopts the pretrained model to a discriminative or generative task by feeding h and parameters W_y into an added linear output layer to predict the label y

$$P(y|x^1, ..., x^m) = \texttt{softmax}(h\ W_y). \tag{2}$$

Then, the following objective is often maximized for supervised fine-tuning as

$$L_2(\mathcal{C}) = \sum_{x,y} \log\ P(y|x^1, ..., x^m) \tag{3}$$

where $\mathcal{C}$ is a labeled dataset. The fine-tuning objective that includes the language modeling objective typically improves generalization of the supervised model and accelerate convergence [76], [77]. Therefore, one of the most efficient fine-tuning objective can be considered a linear combination of the language modeling objective (1) as

$$L_3(\mathcal{C}) = L_2(\mathcal{C}) + \lambda * L_1(\mathcal{C}) \tag{4}$$

where λ is the weight parameter [73].

The GPT-based language understanding system is typically designed to tackle either discriminative or generative tasks, and its capability to generate human-like responses is limited. To enhance conversational models, multiple approaches are often integrated. One effective method involves applying reinforcement learning from human feedback (RLHF) after initially fine-tuning the model on instruction-following tasks [2]. This process includes using demonstration data consisting of desired response texts provided by humans in response to prompts. A reward model is then trained using the sampled human-ranked outputs. Subsequently, a proximal policy optimization algorithm [78] is used to refine the policy based on the rewards determined by the reward model.

For language generation, the process begins with a sequence of input text tokens, which includes specific *start* and *end* tokens. These tokens are crucial for generating the embedding features needed for the language generation process.

The next section reviews several recent approaches based on LLM ensemble techniques for overcoming the limitations of the single-LLM-based approach described in Section I and improving language generation. The review includes some mathematical details that are uncommon in the general ML literature but essential for using these models in language generation.

III. Primary Methods for LLM Ensembling

Since 2022, techniques for ensemble learning with LLMs have been effectively used to model language features. These applications include extracting product features for e-commerce [42], generating image captions [79], and classifying medical diagnoses [80]. One major benefit of ensemble learning is its ability to generalize diverse outputs by combining knowledge through different frameworks. This can be done at the architecture level, such as weight merging and knowledge fusion, or at the model level, like output ensemble and routing (Figure 3). Language generation of LLMs is fundamentally a next-token prediction task. The goal of ensemble language modeling is to improve the quality of predicted tokens. In this section, we review basic methods from the perspective of ensemble techniques.

A. Architecture-Level Ensemble

1) Weight Merging: Weight merging refers to the method of combining the weights of multiple trained models into a single model. This method requires architectural modification and can model the diversity among a set of transformer weights using multiple LLMs.

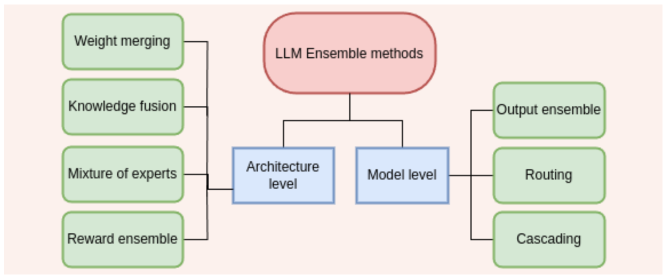

Fig. 3. LLM ensemble methods in our categorization. We identify seven primary approaches at the architecture or model level.

In traditional weight merging, weights from K LLMs are merged where the merge function over the weights W_i for each LLM is typically given by averaging as

$$\mathrm{W} = \frac{1}{K}\sum_{i=1}^{K}\mathrm{W}_i \tag{5}$$

where W is the final model weight. Figure 4 shows weight merging using three source LLMs. K can be simply the number of all LLMs [28] or the top-K LLMs can be determined e.g., by greedy algorithm [18]. W can be further interpolated using pre-trained weights and a parameter α [19]. These weight averaging methods, interpolating between models, are shown to improve accuracy on the fine-tuning tasks.

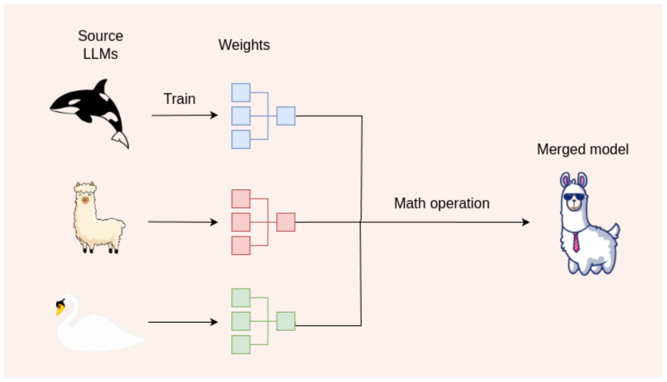

Fig. 4. Weight merging with three source code of LLMs. Weights are typically operated after training models.

Building on those findings, the extended merging methods extrapolate weights between models [29], [30]. In this framework, a task vector T is typically defined as

$$T = W_{ft} - W_{pre} \tag{6}$$

where W_{pre} is the weights of a pre-trained LLM, W_{ft} is the weights of the same model after fine-tuning, and T is transformed and applied to merge weights from multiple LLMs. Given a set of LLMs, T can be transformed based on the formula T_{new} by various network operation techniques, such as arithmetic operation [29], trim, elect, sign [31], random drop and re-scale [32]. Examples of the arithmetic operations are illustrated in Figure 5. The merge function over the task vector T_{new} can then be calculated as

$$\mathrm{W} = \mathrm{W}_{pre} + \lambda\, T_{new} \tag{7}$$

where W_{pre} is the weights of a base pre-trained LLM and the scaling term λ is determined using held-out validation sets.

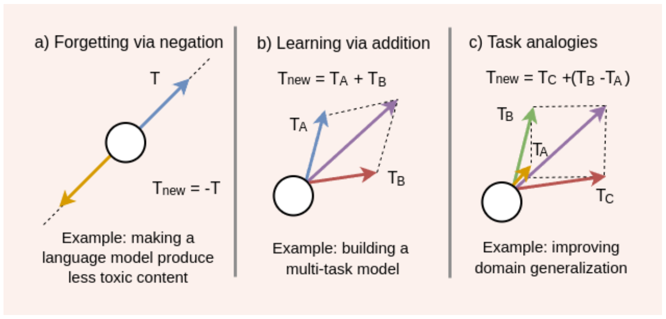
a) Forgetting via negation
T
Tnew = -T
Example: making a language model produce less toxic content
b) Learning via addition
Tnew = TA + TB
TA
TB
Example: building a multi-task model
c) Task analogies
Tnew = TC +(TB -TA)
TB
TA
TC
Example: improving domain generalization

Fig. 5. Arithmetic operations with a task vector T. T is edited to improve model performance [29].

Another approach of perplexity-based merging can be found in [33]. This employs a test-time aggregation approach using importance weights to combine the output logits of LLMs during inference; the perplexity over the input prompt was first validated to measure each LLM's expertise, and then a greedy algorithm was used for perplexity minimization [33] to derive importance weights. This approach extended the weight merging approach to the one at test time. Thus, this training-free ensemble model can easily include newly released open-source LLMs. Because the application of model merging strategies is an expanding area of research, a comprehensive, open-source library has been developed to support them by MergeKit [34].

2) *Knowledge Fusion:* Knowledge fusion is the architecture-level technique to represent the collective knowledge of LLMs by defining different fusing functions and computing the probabilistic distribution matrices of source LLMs [20]. The fusing process of three source LLMs to train a target LLM is shown in Figure 6. This method extends the knowledge distillation technique [81] widely used for text classification, where it transfers knowledge from a larger, more complex model (teacher) to a smaller, simpler model (student) to create an efficient student model.

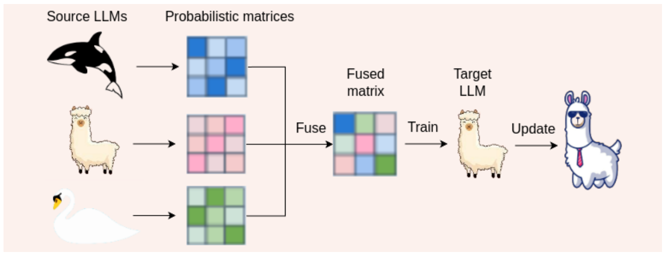

Fig. 6. Fusing three LLMs. Fused knowledge is used to train a target LLM.

The problem setting in knowledge fusion is that improve a target LLM by training with a fused matrix $\mathbf{P}_q$. Given a set of probabilistic distribution matrices $\{\mathbf{P}_q\}_{j=1}^{K}$ of K LLMs for text sample q from the corpus C, $\mathbf{P}_q$ is obtained by

$$\mathbf{P}_q = \mathrm{Fusion}(\mathbf{P}_q^{\theta_1},\ \mathbf{P}_q^{\theta_2}, ...,\ \mathbf{P}_q^{\theta_K}) \tag{8}$$

where θ_j is the parameters of the jth LLM and $\mathbf{P}_q^{\theta_j}$ is made by aggregating token-level predictions. Fusion$(\cdot)$ can be selected among various strategies, e.g., outputs $\mathbf{P}_q$ using the minimum or weighted average of cross-entropy scores [20]. While training a target LLM using the fusion algorithm, the objective function for transferring the capabilities of source LLMs to the target LLM is derived as

$$\begin{aligned} L &= \lambda\ L_{Target} + (1-\lambda)\ L_{Fusion} \\ L_{Target} &= -\mathbb{E}_{q\sim C}\ [\mathbb{D}(\mathbf{Q}_q,\ \mathbf{O}_q)] \\ L_{Fusion} &= -\mathbb{E}_{q\sim C}\ [\mathbb{D}(\mathbf{Q}_q,\ \mathbf{P}_q)] \end{aligned} \tag{9}$$

where $\mathbb{D}(\cdot)$ is the discrepancy function between two matrices (e.g., KL divergence), $\mathbf{Q}_q$ is the output distribution matrix of the target LLM, $\mathbf{O}_q$ is the one-hot label matrix, and λ is the weight parameter.

For LLMs of different size than the target LLM, this knowledge fusion method can be extended to the pairwise method. For each LLM, fixing as a pivot, one can compute pairwise matrices between the pivot LLM and each of the rest source LLMs [44]. The pairwise fused matrix, e.g., $\mathbf{P}_q^j$ can be derived using the pairwise knowledge fusion function as

$$\mathbf{P}_q^j = \text{Fusion}(\mathbf{P}_q^{\theta_v},\ \mathbf{P}_q^{\theta_j})\ |_{v\neq j}. \tag{10}$$

The pairwise fusion objective for each target LLM is then defined as

$$L_{Fusion} = -\mathbb{E}_{q\sim C}\ [\mathbb{D}(\mathbf{Q}_q,\ \mathbf{P}_q^j)] \tag{11}$$

where L_{Fusion} is substituted to L in formula (9) to compute the final objective function.

Different LLMs may use different tokenizers, which can result in varied size of their vocabulary matrices. For instance, LLM1's tokenizer might produce the token "immediate," wheres LLM2's tokenizer generates "immediately." This difference in their vocabulary matrices hinders the mathematical fusion process, known as the token alignment problem [82].

To solve this issue, we can use dynamic programming. This technique helps minimize the total cost of editing one sequence of tokens to match another, using a one-to-one exact mapping [82]. Alternatively, we can use a minimum edit distance approach. This method aligns tokens from different tokenizers based on the minimum number of edits needed, which helps maintain important information in the vocabulary distribution [20].

Another way to tackle the token alignment problem is through an embedding representation approach. This method combines collaboration weights with relative representation matrices from the LLMs. These matrices are based on the similarities between tokens, using anchor tokens for reference. By merging these matrices with collaborative model weights, we improve the model's ability to generalize, making it more effective in tasks like word completion [83], [84].

3) Mixture of Experts: Architecture-level ensemble methods are used in transformers as expert networks. These methods help model the distributions of multidomain data for different tasks. Some of these tasks include text generation, mathematics, code generation, and natural language understanding (NLU) [6], [21], [64].

Given input token x and K expert networks, the output next-token y of an expert layer is typically determined by the weighted sum of the outputs for each expert network $E_i(x)$, where the weights are given by the output of a gating network (router) $G(x)_i$, as shown in Figure 7. This relationship can be expressed mathematically as

$$\text{y} = \sum_{i=1}^{K} G(x)_i \cdot E_i(x) \tag{12}$$

which shows that this model can be considered a mixture of experts (MoEs).

MoEs combine different expert distributions from various domain datasets using a gating mechanism. This approach allows MoEs to create more optimal distributions than individual experts [21]. They scale model sizes more effectively without a significant increase in computational complexity [85].

In the equation 12, the elements of $G(x)_i$ represent the top n experts, where $n > 1$. This selection ensures that the gating functions receive meaningful gradients [12]. The elements in $E_i(x)$ represent the n expert sub-blocks, which may use different sets of weights, such as those in the SwiGLU architecture [86].

As the number of experts per token, n, increases, the model's parameter count rises while keeping computational costs constant. This is because the computation in MoEs is largely sparse [21]. Alternatively, MoEs can be adapted to Switch Routing, where only one expert is selected from a reduced batch size of parallel experts [6]. This $n = 1$ routing strategy simplifies implementation and reduces routing computation and communication costs.

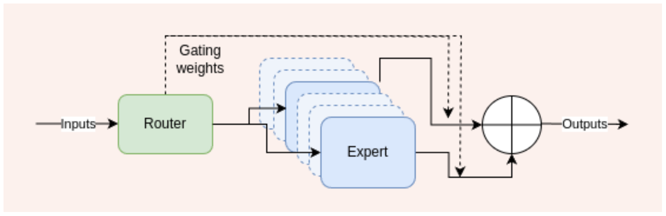

Fig. 7. Mixture of experts layer. Each input token is typically routed to suitable experts to predict a next token.

4) Reward Ensemble: Architecture-level ensembling can be applied to reward modeling in reinforcement learning by using multiple reward models (RMs), known as a reward ensemble. This section discusses the reward ensemble within the context of reinforcement learning from human feedback (RLHF).

This approach was first introduced to address reward hacking, also known as overoptimization [22]. Reward hacking occurs when a policy gains high rewards while ignoring true human preferences [87]. The typical structure of the reward ensemble in the RLHF pipeline is shown in Figure 8.

In this model, the learned RMs aim to approximate human preferences, represented by the policy π. The policy π is optimized in relation to the policy model and the RMs. This can be done using LLMs with different initialization seeds [22] or through low-rank adaptation [48] while supervised fine-tuning (SFT) or pretraining [50].

The initial policy, denoted as π_{init}, and the optimized policy π together create the gold RM score R. This score is calculated based on the Kullback–Leibler divergence $D_{KL}(\pi\ ||\ \pi_{init})$, which varies depending on the optimization method used. If $R(0) := 0$ by convention and α and β are parameters, a reward function of RM can be written as

$$R(d) = d(\alpha - \beta \log d) \tag{13}$$

where $d := \sqrt{D_{KL}(\pi \mid\mid \pi_{init})}$. When using Mean Optimization method [88], the ensemble reward function of R given K LLMs can be written as

$$R(q, a) := \frac{1}{K} \sum_{i=1}^{K} R_i(q, a) \tag{14}$$

where R_i is the gold RM score given the prompt q and its response a sampled from i-th RM. In addition to this method, we can test various optimization techniques based on performance criteria. Examples include Worst-Case Optimization [88] and Uncertainty-Weighted Optimization [89].

Since computing RM ensembling is costly, a multi-head RM approach has been suggested. This method saves memory and time when training LLMs by using shared encoders with separate linear heads [49].

We can also extend the RM ensemble to include weight-averaged reward models using (5) and fine-tune multiple RMs with different hyperparameters. This strategy enhances scalability, improves robustness against label corruption, and increases reliability during distribution shifts while mitigating reward hacking [47].

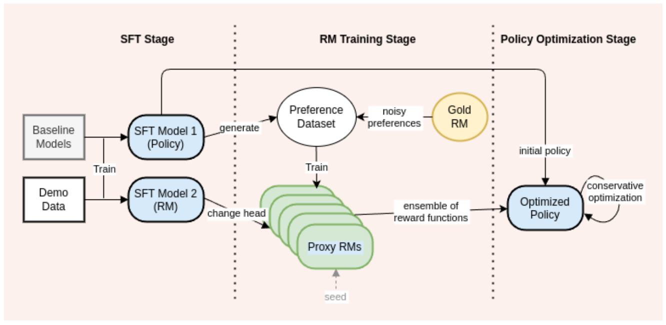

Fig. 8. RLHF pipeline in reward ensemble. Multiple RMs are used to mitigate reward hacking.

B. Model-Level Ensemble

1) Output Ensemble: Output ensemble is a technique that does not require training LLMs. It combines responses from several LLMs to enhance performance, accuracy, and robustness compared to using a single LLM.

In a classification task, output ensemble can involve voting or classifying outputs from multiple LLM agents. Voting means choosing the most common output from the responses of different LLMs. The final output label $\mathbf{y}$ of majority voting over the LLMs' outputs can be written as

$$\mathbf{y} \sim \#argmax \ \{i \mid y_i = \mathbf{y}\} \tag{15}$$

where y_i is the output label by i-th agent. In classifying outputs, a fully connected neural network can be trained to ingest the responses of all LLMs to produce a class prediction [52].

For a text generation task, output ensemble can be applied to combine output texts by prompting an LLM to synthesize them. Given an input prompt x, the synthesized text y over the n LLM outputs can be written as

$$y = \oplus_{j=1}^{n} [A_j(x)] + x \tag{16}$$

where $A_j(x)$ is the output text from j-th LLM, $\oplus_{j=1}^{n}[A_j(x)]$ represents application of the synthesize prompt for fusing n output texts, and $+$ means concatenation of texts [23]. The formula 16 represents the process of generating and synthesizing responses from n LLMs.

The output ensemble has been effectively used to rank and fuse the top-K generated texts. This approach takes advantage of the strengths of each base LLM. Research shows that this post-ranking fusion framework can better capture the differences between agent responses [53]. It also produces better results by utilizing the top-K outputs [53], [54].

In this framework, various transformer-based ranking models can be trained and tested using the CE criterion. Examples include PAIRRANKER [53], URG [54], and SummaReranker [90]. These ranking models typically use a scoring system to evaluate and rank the outputs, leading to a sorted list of results.

- *Scoring model*:
  A scoring model produces meaningful numeric values by either classifying data with an ML model or generating results with a language model. These models are commonly used in ranking and various text generation tasks, such as outputs ranking, instruction-following [53], QA reasoning [55], and image caption generation [79].
  For instance, a random forest (RF) classifier can be trained to give confidence scores for answers generated by LLMs. This helps in choosing the most confident response [55].

Additionally, scoring models are important for routing tasks.

2) Routing: Routing is a technique that directs input queries to multiple LLMs. The design of a multi-LLM router is illustrated in Figure 9. In this system, the router assigns a score to each input query using a scoring model. It then sends the query to the most appropriate LLM based on that score. The scoring function varies depending on the design and the specific tasks.

For general tasks like QA, summarization, and information extraction, a BART score [91] can evaluate the quality of responses from different LLMs. This score is effective because it correlates well with the actual answers [59].

For tasks involving mathematical or natural language reasoning, a confidence score from a transformer-based classifier, such as RoBERTa or T5, can help identify the most reliable LLM to use [60].

This classifier-based routing can be extended to different fields by training multiple classifiers. For instance, in finance, legal, and general knowledge tasks, the output from a kNN classifier $g_m(\cdot)$ can serve as a correctness score for the LLM:

$$g_m(x_i^d) = \frac{1}{k} \sum_{e \ \in \ \text{NN}(\ \phi(x_i^d), \ k, \ D)} y \ (e, m) \tag{17}$$

where x_i^d is a sample from a domain dataset d, $y \ (e, m) \in \{0, 1\}$ is the correctness of model m on the embedded input

e, and NN($\phi(x_i^d)$, k, D) is the set of k closest embedded neighbors from D datasets to the new embedded sample $\phi(x_i^d)$ [57]. x_i^d can be embed using an embedding model, e.g., sentence transformer ϕ [92].

This classifier-based router can also incorporate the cost-performance trade-off [24], [65]. For example, the kNN router in (17) can model a performance score using the willingness-to-pay parameter λ

$$\text{performance score}_{i,j} = \lambda \cdot P_{i,j} - \text{cost}_j \tag{18}$$

where the kNN classifier estimate the predicted performance P_{ij} of j-th LLM on sample x_i, cost is the total cost approximated using the cost per token metric, and higher λ indicates a preference for superior performance at a higher cost [24].

This routing technique can also improve the reward ensemble method for RLHF in Section III-A4 by sampling from a routed LLM. This allows for training a single RM instead of multiple RMs [58].

Routers can be cost-effective models that simplify the process of connecting inputs to outputs. However, they struggle to save costs when handling complex reasoning tasks. For example, in mathematics, routers determine the difficulty of questions and the correctness of answers based only on their text descriptions. This reliance can lead to lower accuracy [9]. Additionally, fine-tuning routers using smaller language models may require them to learn from a large number of training samples, which can complicate the learning process.

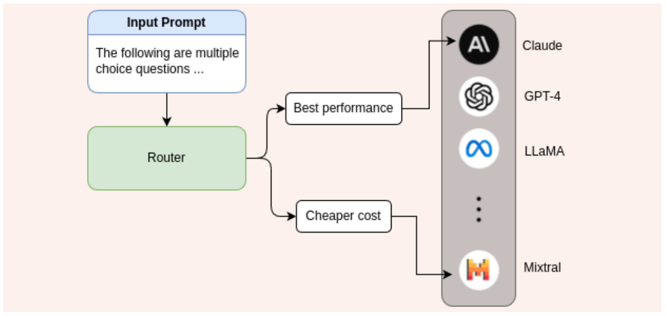

Fig. 9. Routing system with LLMs of different cost and performance. It routes the input query to a corresponding LLM depending on preferences.

3) Cascading: To address the routing problems mentioned above, cascading techniques have been proposed. In this approach, a list of LLM APIs is called one after the other. A decision maker then chooses whether to use a response from a weaker LLM or a stronger one [62]. Figure 10 illustrates the typical cascade process, which involves chaining three LLMs. This includes a weaker, more affordable LLM and stronger, more expensive ones.

In this cascade, the weaker LLM is used first to generate an initial answer along with some metadata. This answer is then given to a decision maker. The decision maker evaluates whether the answer can be accepted as final, based on a score and a predetermined threshold. For instance, in mathematical reasoning tasks, an agreement score s can measure the consistency of the j-th LLM:

$$s = \frac{\sum_{i=1}^{K} \mathbb{1}_{A_i^j = A^j}}{K} \tag{19}$$

where A^j is the answer from j-th LLM. If s is higher than a threshold, A^j is selected as the most consistent final answer among K samples [9]. If the system rejects the initial answer, it uses a more powerful LLM to find a better response. At the same time, it calculates the total cost of answering the question. This step-by-step approach can deliver performance similar to that of a stronger LLM compared to the one-step process in routing. It also helps to significantly lower costs.

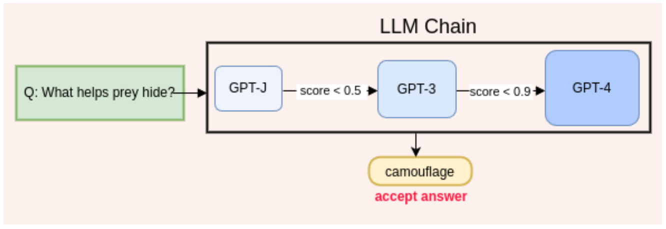

Fig. 10. The cascade chain of three LLMs. Getting acceptable answers from weaker but cheaper LLMs can save the cost of invoking stronger costly LLMs.

C. *Prompt Engineering*

Prompt engineering is a technique that improves the quality of output from a single LLM. Key methods include prompt ensemble and prompt augmentation. Prompt engineering is not an LLM ensemble technique, but we discuss some relevant approaches, as they can enhance the LLM ensemble results.

Prompt ensemble combines results from multiple prompts. In text generation tasks like multiple-choice (MC) questions, the sensitivity of LLMs to prompts [93] can be leveraged to create a diverse set of prompts. This diversity helps improve performance by paraphrasing templates and permuting examples. For a MC question with n prompts, the predictive distributions from these prompts are averaged to determine the ensemble accuracy [5].

Prompt augmentation involves adding relevant information to a prompt to boost reasoning ability. A well-known method is chain-of-thought prompting (CoT). CoT breaks down complex problems into smaller steps, providing examples for each step before arriving at the final answer [94]. An example of a CoT prompt with one intermediate step is illustrated in Figure 11. Other methods, like program-of-thought prompting (PoT) [95] and retrieval-augmented prompting (RA) [96], also enhance output quality.

LLMs with smaller parameter sizes can experience various performance issues, such as problems with semantic understanding, missing steps, and errors in symbol mapping. Therefore, LLMs should have a large number of parameters, ideally over 540 billion. Performance then improves when the LLM learns from a few input-output examples.

To help LLMs tackle problems harder than examples, an extension of CoT called least-to-most prompting (LtM) has been developed [97]. This method solves sub-problems sequentially and augments the answers to the prompt in a divide-and-conquer approach. Compared to CoT, LtM often allows LLMs to generalize better on complex reasoning tasks.

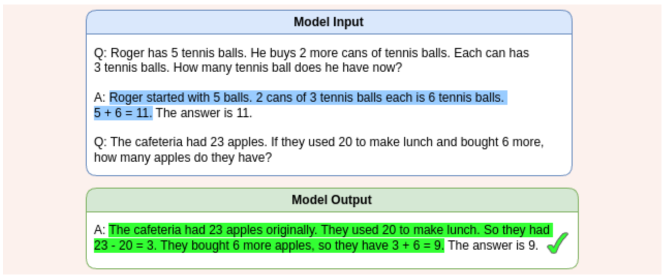
Model Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis ball does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9.

Fig. 11. The input prompt of CoT with one intermediate step exemplar [94].

IV. Text and Code Generation using LLM Ensemble Methods

We have seen success in using LLM ensembles for various language understanding tasks. We believe this approach can also be applied to real-world text and code generation tasks. Our goal is to expand the use of LLM ensemble models in these areas while overcoming the limitations associated with single-LLM approaches mentioned earlier.

LLM ensemble learning is a relatively new research topic. Until recently, its application to text and code generation had not been thoroughly explored. As the techniques have developed, several articles have been published on related topics such as instruction following [23], [53], [57]–[59], [64], reasoning [6], [9], [55], MMLU [61], and code generation [21], [24].

We selected these articles based on their performance results and potential for wider applications. The findings show that LLM ensembling techniques enhance the naturalness, accuracy, and quality of generated outputs. We categorize these LLM ensemble approaches into four methods based on their definitions in Section III. This section will discuss the model structure, training methods, subjective evaluation, and benefits of these approaches.

A. Code Generation Using MoE Ensemble Models

In this approach, the MoE ensemble technique is used for input-to-expert routing in language generation. Input-to-expert routing identifies which experts to use based on the input tokens. It builds on traditional methods, such as gating networks in RNN-based machine translation [12] and NN-based vowel discrimination [98].

One example of this method is transformer-based code generation using MoEs. This approach improves on conventional LLM training [21]. It achieves two key improvements: First, it uses only 13 billion active parameters per token, outperforming larger models like Llama2, which uses 70 billion parameters per token. Second, the MoE layer in Figure 7 replaces the FFN sub-block in the transformer block, allowing for better learning of language features. To speed up execution on single GPUs, Megablocks [99] casts the FFN operations of the MoE layer as large sparse matrix multiplications. As for its training and implementation, remind Section III-A3.

We conducted subjective evaluations to demonstrate the effectiveness of this approach [21]. The results are summarized in Table I, which shows the pass rates from a code functional correctness test on three models. For instance, 60.7% of the code generated by the Mixtral model passed the unit tests, outperforming the LLaMA model. The LLaMA 2 70B and GPT-3.5 models were built using transformers with 70 billion and 175 billion parameters, respectively. In contrast, the Mixtral model uses a transformer with 8 experts and 47 billion parameters, effectively modeling sparse mixture of expert networks at each layer of a decoder-only model. The table shows that Mixtral, with fewer parameters, achieved significantly better functional correctness than a single LLM with many more parameters.

This approach is not limited to code generation; it can also be applied to text generation tasks. See Table IV for a list of applications areas of LLM ensemble models. For example, it can be used for instruction-following with the MoE transformer [64] and for common sense reasoning with Switch transformers [6]. Similar to the code generation method, input-to-expert routing is determined by the weighted outputs of the mixture experts in a gating network based on the input tokens. MoEs are then used to model the expert function between source and target tokens for each expert. The subjective evaluation results confirm the effectiveness of these approaches when using natural language tokens as features. Notably, for Switch transformers, a time-to-quality threshold score improved from 131.1 to 62.8 with the optimized architecture [6].

TABLE I
The subjective pass rate (pass@1) among code generated using the LLaMA 2 70B, GPT-3.5, and Mixtral 8x7B models [21]

Dataset	LLaMA 2 70B	GPT-3.5	Mixtral 8x7B
MBPP	49.8%	52.2%	**60.7%**
HumanEval	29.3%	-	**40.2%**

Mixtral with fewer parameters achieved significantly better functional correctness than the use of a single LLM with a lot more parameters.

B. Text Generation Using Output Ensemble Model

This approach uses an output ensemble model to improve text generation by combining a ranker and a fuser. For instance, a post-ranking fusion framework with two separate language models has been proposed [53]. In this framework, the ranker assesses linguistic quality, while the fuser merges the top-ranked candidate outputs.

First, the input prompt x is combined with the responses from K LLMs using separator tokens to create the input data. Then, the generative model is trained to produce a final response y^* by fusing the top-ranked responses (Figure 12).

Specifically, the ranker's input features include the instruction and a set of N LLM output responses in English, formatted as a JavaScript array. The fuser's input consists of the top K responses from N LLMs, which it uses to generate the final output. Each training sample is represented by a language feature vector, which includes multiple frames of text embedding (e.g., a 768-dimensional vector of floating-point numbers) and quality scores. A single transformer learns conversational text features, allowing to model the linguistic quality score between the instruction and response features. The ranker and fuser are trained separately in this framework.

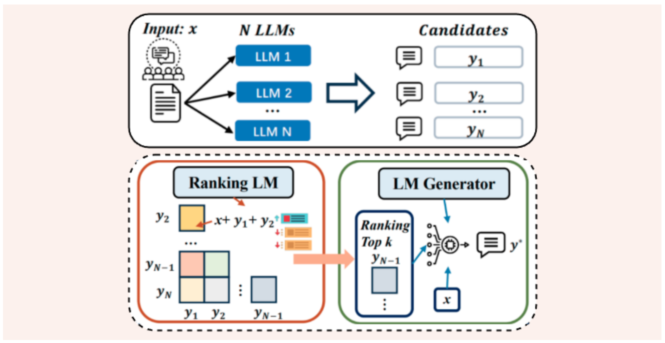

Fig. 12. Post-ranking fusion framework using output ensemble applied to an instruction-following task. It consists of a ranker and a fuser in 2 steps [53].

To handle the different tasks of score prediction (creating a ranked list of texts) and text generation (producing the fused text), the approach employs a cross-attention transformer for the ranker and a fine-tuned T5 model [71] for the fuser. The transformer includes a multi-layer perceptron that takes encoded chat texts as input and candidates in a single-head layer. Training for the transformer ranker starts with supervised learning on the instruction dataset. Here, a multi-layer perceptron is trained using chat text features as observations. Transformer features are derived from the text embedding data, and various scoring functions, such as BERTScore [100] and BARTScore [91], are used to optimize quality scores. This is followed by supervised fine-tuning of the T5 fuser, where the input and K output texts are concatenated using separator tokens to generate an enhanced output. This step models the relationship between the input instruction and each candidate response using a sequence-to-sequence prediction approach.

This training method has two main advantages over using a single LLM for generation: 1) Each component is flexible and can be adapted for different applications. For example, it can be extended for code generation by using a listwise ranker to improve output. Instead of ranking candidate responses against the input instruction, as in instruction-following tasks, a code generation task can train a listwise ranker [101] on public code datasets [102] to rank multiple code snippets by runtime, potentially reducing the runtime of the fused code. 2) The strong output from a closed-source LLM, like ChatGPT, can be improved through post-ranking fusion. This differs from architecture-level ensemble generation, which only use open-source LLMs. Since a powerful LLM can give a solid baseline, functional correctness of the fused code can be ensured.

Table II presents the natural language generation (NLG) scores for three instruction-finetuned LLMs: the T5 baseline (Flan-T5), the LLaMA2 baseline (Alpaca), and the output ensemble model using the proposed post-ranking fusion approach (LLM-BLENDER) [53]. Comparing LLM-BLENDER with Alpaca and Flan-T5 shows that the proposed method outperforms traditional single LLM baselines in modeling and generating conversational texts. The quality improvement from the baselines to LLM-BLENDER indicates that the strengths of multiple LLMs are effectively captured when diverse responses are combined.

TABLE II
THE SUBJECTIVE NLG SCORES AMONG RESPONSE TEXTS GENERATED USING THE LLM-BLENDER, ALPACA, AND FLAN-T5 MODELS [53]

Model	BERTScore	BARTScore	BLEURT
LLM-BLENDER	**79.09**	**-3.02**	**-0.17**
Alpaca	71.46	-3.57	-0.53
Flan-T5	64.92	-4.57	-1.23

LLM-BLENDER ensembling open-source LLMs outperforms the conventional single LLM baselines by improving conversational text quality.

C. Results Aggregation Using Output Ensemble Models

This approach, like the previous one, generates response texts based on input instructions using a training-free output ensemble model. However, instead of fine-tuning a language model to combine the top-K LLM responses, this method prompts an LLM to synthesize responses generated from pre-selected LLMs.

A Mixture-of-Agent (MoA) approach has been proposed for text generation [23]. The MoA consists of multiple layers of LLM agents. Each layer updates the outputs from the previous layer to enhance performance. Figure 13 shows the structural model representation for a four-layer MoA. The top three layers of the MoA form a feed-forward structure that integrates responses in a left-to-right direction using the Aggregate-and-Synthesize prompt (Figure 14).

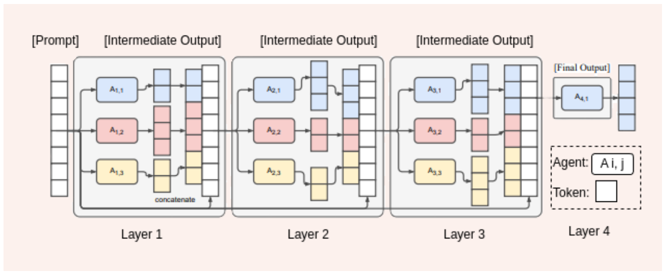

Fig. 13. MoA structure with 4 layers with 3 LLM agents in each layer [23].

You have been provided with a set of responses from various open-source models to the latest user query. Your task is to synthesize these responses into a single, high-quality response. It is crucial to critically evaluate the information provided in these responses, recognizing that some of it may be biased or incorrect. Your response should not simply replicate the given answers but should offer a refined, accurate, and comprehensive reply to the instruction. Ensure your response is well-structured, coherent, and adheres to the highest standards of accuracy and reliability.

Responses from models:
1. [Model Response from $A_{i,1}$]
2. [Model Response from $A_{i,2}$]
...
n. [Model Response from $A_{i,n}$]

Fig. 14. Aggregate-and-Synthesize prompt to integrate multi-response [23].

Each LLM first produces a response text to provide more context and diverse perspectives. Similar to the MoE-based approach discussed in Section IV-A, the MoA combines the roles of the gating network and expert networks, making it a model-level MoE. During synthesis, responses are refined iteratively by re-aggregating them at each layer. The responses from the last layer are fed to the aggregator LLM, which uses the Aggregate-and-Synthesize prompt to generate a single high-quality output.

This output ensemble method could also enhance code generation quality by addressing coding problems using the MoA structure and prompt augmentation techniques discussed in Section III-C. For example, we can prompt programming questions to generate solutions in Python, C++, Java, JavaScript, and Go and test their functional correctness using HumanEval dataset [1]. A subjective instruction-following accuracy test compared the MoA-based model's performance with the latest LLM [23]. The results are shown in Table III. In this experiment, the latest LLM ranked first on the leaderboard as of July 22, 2024. AlpacaEval 2.0 and MT-Bench served as benchmarks for instruction-following tasks, evaluating models based on a nuanced understanding of correctness, relevance, and efficiency. The table shows that the MoA-based ensemble model with deeper layers achieved better quality than the latest single LLM according to human-validated accuracy scores.

Other approaches of output-ensemble-based QA reasoning can be found [55]. One alternative [55] combined output ensemble and prompt ensemble methods. It used CoT or RA prompts as inputs for output-ensemble-based QA reasoning. This QA prediction did not require prior knowledge of the question's reasoning type; it only used a scoring classifier, making it more computationally cost-effective than MoA.

The LLM ensemble approach with output ensemble models has also been applied to other tasks, such as clinical text error detection [103], disease diagnoses [56], [104], and image retrieval [79]. A prompt-based results aggregation method was proposed for clinical text error detection [103]. In this method, subtasks of error detection and correction were all asked in a single prompt, aggregating results from the latest closed-source LLMs. A majority-vote method [56] and a frequency-based weighted method [104] have also been used in this prompt-based aggregation. In one study [56], JSON was utilized in CoT-augmented prompts to model nested responses and graph-like structures.

TABLE III
THE SUBJECTIVE INSTRUCTION-FOLLOWING ACCURACY SCORES AMONG RESPONSE TEXTS GENERATED USING MOA AND MOA-LITE CORRESPOND TO THE 6 PROPOSERS WITH 3 LAYERS AND WITH 2 LAYER RESPECTIVELY, AND GPT-4O MINI [23]

Model	AlpacaEval2.0 LC Win.	MT-Bench Avg.
MoA	**65.1**	**9.25**
MoA-Lite	59.3	9.18
GPT-4o mini (05/13)	57.5	9.19

MoA-based open-source ensemble model with a deeper layer outperforms the single state-of-the-art closed-source LLM in instruction-following tasks.

D. Cost Effective Text Generation Using Routing Models

This approach uses closed-source LLMs to achieve a cost-efficient performance that is 40% cheaper than that of a stronger LLM. It does this by modeling mixture-of-thought (MoT) representations.

A cascade-based language reasoning method was proposed in [9]. In this method, routing decisions are made based on the consistency of answers from a weaker LLM [105]. As explained in Section III-B, an LLM cascade involves sequentially calling LLM APIs with prompts using a router, as illustrated in Figure 10.

To check for consistency, answers are first sampled from the weaker LLM using MoT representations. Each prompt includes CoT and PoT prompts, with eight examples for each type. This setup allows for 8-shot in-context learning. The consistency of the weaker LLM's answers is assessed through voting or verification. The router uses this consistency score to decide whether to accept or reject the LLM's answer. We evaluate both task accuracy and cost efficiency to understand their relationship.

A subjective reasoning test compared the performance of cascade-based models with stronger LLMs [9]. Figure 15 shows the cost-effective performance curves for these models. In this experiment, GPT models using CoT or PoT prompts and cascade-based models were compared. The results indicate that the cascade-based MoT model performed similarly to the GPT-4 model with CoT prompts, but at half the cost. Since coding requires reasoning skills, this cascade-based routing method could also generate optimal code at a lower cost.

The LLM routing approach, which combines weaker and stronger LLMs, can be applied to other text generation tasks, such as instruction-following [57]–[59], code generation [24], and MMLU [61]. For instruction-following tasks, routing-based methods have been developed to select the best output from various LLMs using reward models [58] or classifiers [57], [59]. In code generation and MMLU tasks, routing-based cost-effective methods have been benchmarked [24], [61]. The subjective evaluation results demonstrate the efficiency of this routing approach, achieving near-optimal performance at a low cost.

E. Comparisons Among These Four Approaches

The MoE approach with transformer models resembles traditional RNN-based language models. It maintains the input-to-expert routing step, requiring only minor changes to the existing language generation architecture for offline training [21]. Output ensemble methods, such as post-ranking fusion models or MoA models, expand a single LLM response into multiple responses [23], [53]. This allows for better capture of diversified long-range dependencies and improved generalization compared to conventional single LLM approaches. While training-based text generation from transformers can be complex [53], prompting LLMs for responses is simpler [23]. However, the computational time for LLMs can be slow due to the feed-forward multiLLM generation process with CoT prompting. Table IV summarizes these recently proposed text and code generation approaches using LLM ensemble techniques. The model structure and advantage columns in the table also summarize different research directions. We analyze these approaches from three aspects based on their modeling traits and the relationship between input and output features. Further discussions on these approaches will be found in the ”Discussion” section.

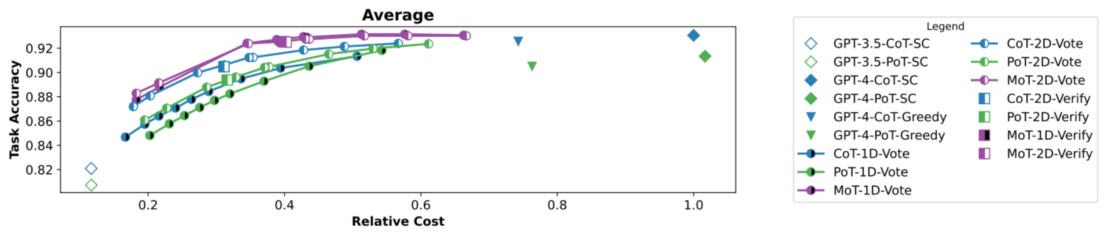

Fig. 15. The performance curves of cost vs. accuracy of reasoning answer using (blue) the GPT with CoT models, (green) the GPT with PoT models, and (purple) the MoT cascade models [9]. The MoT cascade model performed similarly to the GPT-4 model with CoT prompts at half the cost in reasoning tasks.

TABLE IV
A SUMMARY OF THE PROPOSED TEXT AND CODE GENERATION APPROACHES USING LLM ENSEMBLE TECHNIQUES

LLM Ensemble Method	Application	Model Structure	Advantage	Generated Features
A. MoE	Code generation	MoE transformer [21]	Parameter efficient open-source LLM	Next code token
	Instruction-following	MoE transformer [64]		Next text token
	Reasoning	Switch transformer [6]		Next text token
B. Output ensemble	Instruction-following	Ranker and fuser transformers [53]	Extension flexible component	Enhanced text tokens
C. Output ensemble with MoA	Instruction-following	Feed-forward structure [23]	Training-free agentic workflow	Synthesized response texts
D. Routing	Reasoning	LLM cascade + Decision maker [9]	Cost effective	Responded texts + Score
	QA reasoning	MoA + RF [55]		Aggregated response texts
	Instruction-following	LLM routing + RL [58]		Responded texts + Score
	Instruction-following	LLM routing + kNN [57]		Responded texts + Score
	Instruction-following	LLM routing + transformer [59]		Responded texts + Score
	Code generation	LLM routing + kNN or MLP [24]		Responded code + Score
	MMLU	LLM routing + Light LLM [61]		Responded texts + Score

Model Structure and Advantage also summarize various research directions.

V. Discussion

A. Input and Target Features

In LLM ensembling for language generation, the input features depend on the model being used. Typically, language token embeddings serve as input features for transformer architectures [6], [21], [53], [64]. In a training-free model-level approach, we can only customize prompt engineering and agent structuring. Therefore, the input features for MoA and LLM cascade models usually consist of text prompts enhanced by prompt augmentation techniques [9], [23], [55].

In contrast, LLM routing models generally use text prompts or code snippets without prompt ensembling [24], [57]–[59], [61]. This suggests that routers should work with the original input to better classify which LLM to use.

Various output features for language generation are listed in Table IV. These features indicate that an effective combination of metrics is necessary, depending on the task and model. Section II explains that LLMs excel at capturing long-range dependencies in language features, which relate to human perception. Some LLM ensemble approaches consider this when evaluating their output text features, hinting at the potential for evaluating output code features as well. Human-perceptual correlations in text can be measured using metrics like BERTScore or BARTScore, comparing generated text to ground truth text [53], [57]. Similarly, LLM ensemble models can be evaluated in code generation by using CodeBERTScore [106] to compare generated code snippets to the ground truth.

In language generation with LLM cascade and routing models, the predicted score plays a crucial role in determining the final response [9], [24], [57]–[59], [61]. The quality of the final response from a routed LLM indicates the performance of this prediction.

B. Model Structures and Model Training

Table IV shows that different model structures are used in these four approaches to achieve various goals. MoEs are employed for gating and expert functions, allowing for input-to-expert routing [6], [21], [64]. In contrast, post-ranking fusion transformers are used to represent input and output text features, enabling multioutput ranking and fusion [53].

In routing-based models, routers help invoke multiple LLMs sequentially [9] or route to the best single LLM [24], [57]–[59], [61]. They aim to produce cost-efficient outputs.

We recommend specific training or scoring options for each approach. The depth of the architecture, or the number of layers, is crucial for feed-forward ensemble models. In MoA-based instruction-following [23], the results in Table III indicate that increasing the number of layers improves the quality of synthetic responses due to more auxiliary information.

In other MoE studies [6], [21], [64], researchers tuned the number of transformer layers to minimize the cross-entropy loss between targets and outputs in training sets. The findings suggest that using more than 12 layers can enhance prediction accuracy. However, the optimal depth for MoE layers is typically not as deep as that used in switch-transformer-based text generation. This is reasonable, as n the number of experts to be routed in switch transformer is 1 while $n > 1$ in MoE.

In routing-based approaches, various scoring strategies have been used. These include agreement scores to measure LLM consistency for reasoning [9], reward scores for instruction-following [58], and scoring methods using kNN, transformers, or LLMs for inference optimization [24], [57], [59], [61]. Given the high computational cost of training MoEs and other transformer models, one option is to skip the training process altogether by using a model-level agent ensemble [23], [55].

C. A Comparison Between Text and Code Generations Both Using LLM Ensemble Models

We discuss how LLM ensemble architectures are used in text generation and how they differ from those used in code generation. Understanding these differences can help us adapt text generation models for code generation tasks.

Transformers are the leading models for text generation. In this approach, a transformer is trained to convert an input sequence from an encoder (which processes natural language tokens) into an output sequence from a decoder. The decoder links the encoder's representations to higher-level linguistic texts during generation.

LLM ensembling is commonly used for text generation. However, similar model structures have also been proposed for code generation, as shown in Table IV. Notably, the MoE and routing-based approaches for code generation [21], [24] use the same model structure as those used for text generation.

One of the key differences between the two is the training data. Text generation uses natural language data, wheres code generation relies on programming language data to fine-tune the LLM's specialization.

Another difference lies in the evaluation metrics. For LLM-ensemble-based text generation, we typically measure instruction-following accuracy, answer correctness, and linguistic quality. In contrast, code generation focuses on functional correctness, runtime, and memory usage. Therefore, the metrics for code generation must reflect the specific requirements for evaluating code snippets. For instance, the pass rate of unit tests is a relevant metric for code generation but is not used in text generation tasks. However, some automatic metrics from text generation, like BERTScore [100] and BLEU [107], can be adapted for code generation, leading to metrics like CodeBERTScore [106] and CodeBLEU [108].

VI. Conclusion

This article reviews new language generation methods that use LLM ensemble learning techniques. Compared to traditional methods that rely on a single GPT-based LLM, ensemble models at the architecture and model levels (e.g., MoEs, MoAs, post-ranking fusion, and routing) offer better representation of diversity among LLMs. They also combine output features more effectively, leading to improved naturalness, similarity to target responses, and overall quality of generated text or code.

We have examined various implementations of LLM ensemble techniques for text and code generation in current literature. To provide a clearer overview, we chose the effective approaches that could be applied in real-world scenarios. We grouped these approaches into four LLM ensemble methods and analyzed each method and connect findings systematically.

The findings reveal four key advantages of using these methods: 1) MoE models are parameter-efficient and open-source. They can outperform larger competitive LLMs with many more parameters. This makes them strong candidates for the future development of LLMs. 2) The output ensemble model utilizes a post-ranking fusion framework, consisting of a stand-alone ranker and fuser. This structure allows for flexibility in various extensions and applications. For instance, a conversational text fuser can be adapted to a code fuser for code generation, or a code ranker can score a list of code snippets. 3) The open-source MoA models have a training-free agentic workflow, making them easy to integrate and deploy. They can be flexibly adapted to different tasks by testing various prompts. 4) Routing provides a cost-effective way to manage powerful but expensive LLMs like GPT-4 and Gemini. This technique can greatly improve scalability, especially when handling thousands of prompts.

Many real-world tasks, such as chatbot assistance, medical diagnosis, mathematical reasoning, and code optimization, involve text or code generation. Therefore, our findings are relevant for model selection across various domains.

Despite the successes of LLM ensemble learning methods in text and code generation discussed in this article, important issues still need to be addressed to fully utilize the intrinsic strengths of LLM ensemble methods. For example, current methods have struggled to model and predict code tokens effectively using architectural LLM ensemble models [20], [32]. Given the differences between coding and natural language, we may need ensemble structures specifically designed for programming language modeling and prediction. Additionally, LLM ensemble approaches have not yet actively considered the generation optimality of multimodal features, including image, video, and acoustic features. We believe a promising direction for future research is to apply LLM ensemble models in a multimodal way. This research direction includes the models with better programming language modeling capabilities, such as abstract syntax trees, to code generation tasks and applying the LLM ensemble methods to multimodal LLMs in training and agentic workflow.

References

[1] OpenAI, "Evaluating large language models trained on code," 2021, arXiv:2107.03374.

[2] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *NeurIPS*, New Orleans, USA, 2022, pp. 27 730–27 744.

[3] Anthropic, "Claude 3.5 sonnet," 2024. [Online]. Available: https://www.anthropic.com/news/claude-3-5-sonnet

[4] T. Lab, "Alpacaeval: A leaderboard for evaluating instruction following models," 2023, accessed: 2023-10-05. [Online]. Available: https://tatsu-lab.github.io/alpaca_eval/

[5] M. Jiang, Y. Ruan, S. Huang, S. Liao, S. Pitis, R. B. Grosse, and J. Ba, "Calibrating language models via augmented prompt ensembles," in *ICML 2023 Workshop on Deployable Generative AI*, Honolulu, USA, 2023.

[6] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *JMLR*, vol. 23, no. 120, pp. 1–39, 2022.

[7] S. Welleck, A. Bertsch, M. Finlayson, H. Schoelkopf, A. Xie, G. Neubig, I. Kulikov, and Z. Harchaoui, "From decoding to meta-generation: Inference-time algorithms for large language models," *TMLR*, 2024.

[8] N. Stiennon, L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano, "Learning to summarize with human feedback," in *NeurIPS*, vol. 33, Canada, 2020, pp. 3008–3021.

[9] M. Yue, J. Zhao, M. Zhang, L. Du, and Z. Yao, "Large language model cascades with mixture of thoughts representations for cost-efficient reasoning," in *ICLR*, WIE, AT, 2024.

[10] Z. Zhao, E. Wallace, S. Feng, D. Klein, and S. Singh, "Calibrate before use: Improving few-shot performance of language models," in *ICML*, vol. 139. PMLR, 2021, pp. 12 697–12 706.

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *NeurIPS*, vol. 33, Canada, 2020, pp. 1877–1901.

[12] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," in *ICLR*, Toulon, France, 2017.

[13] Meta, "The llama 3 herd of models," 2024, arXiv:2407.21783.

[14] OpenAI, "Gpt-4 technical report," 2023, arXiv:2303.08774.

[15] DeepMind, "Gemini: A family of highly capable multimodal models," 2024, arXiv:2312.11805.

[16] Y. Gu, L. Dong, F. Wei, and M. Huang, "MiniLLM: Knowledge distillation of large language models," in *ICLR*, WIE, AT, 2024.

[17] W. Merrill and A. Sabharwal, "The expressive power of transformers with chain of thought," in *ICLR*, WIE, AT, 2024.

[18] M. Wortsman, G. Ilharco, S. Y. Gadre, R. Roelofs, R. Gontijo-Lopes, A. S. Morcos, H. Namkoong, A. Farhadi, Y. Carmon, S. Kornblith, and L. Schmidt, "Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time," in *ICML*, vol. 162. PMLR, 2022, pp. 23 965–23 998.

[19] G. Ilharco, M. Wortsman, S. Y. Gadre, S. Song, H. Hajishirzi, S. Kornblith, A. Farhadi, and L. Schmidt, "Patching open-vocabulary models by interpolating weights," in *NeurIPS*, vol. 35, LA, USA, 2022, pp. 29 262–29 277.

[20] F. Wan, X. Huang, D. Cai, X. Quan, W. Bi, and S. Shi, "Knowledge fusion of large language models," in *ICLR*, WIE, AT, 2024.

[21] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mixtral of experts," 2024, arXiv:2401.04088.

[22] T. Coste, U. Anwar, R. Kirk, and D. Krueger, "Reward model ensembles help mitigate overoptimization," in *ICLR*, WIE, AT, 2024.

[23] J. Wang, J. Wang, B. Athiwaratkun, C. Zhang, and J. Zou, "Mixture-of-agents enhances large language model capabilities," 2024, arXiv:2406.04692.

[24] Q. J. Hu, J. Bieker, X. Li, N. Jiang, B. Keigwin, G. Ranganath, K. Keutzer, and S. K. Upadhyay, "Routerbench: A benchmark for multi-llm routing system," 2024, arXiv:2403.12031.

[25] L. Breiman, "Bagging predictors," *Springer Nat. ML*, vol. 24, no. 2, pp. 123–140, 1996.

[26] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *JCSS*, vol. 55, no. 1, pp. 119–139, 1997.

[27] T. Li, Q. Liu, T. Pang, C. Du, Q. Guo, Y. Liu, and M. Lin, "Purifying large language models by ensembling a small language model," 2024, arXiv:2402.14845.

[28] L. Choshen, E. Venezian, N. Slonim, and Y. Katz, "Fusing finetuned models for better pretraining," 2022, arXiv:2204.03044.

[29] G. Ilharco, M. T. Ribeiro, M. Wortsman, L. Schmidt, H. Hajishirzi, and A. Farhadi, "Editing models with task arithmetic," in *ICLR*, 2023.

[30] E. Yang, Z. Wang, L. Shen, S. Liu, G. Guo, X. Wang, and D. Tao, "Adamerging: Adaptive model merging for multi-task learning," in *ICLR*, WIE, AT, 2024.

[31] P. Yadav, D. Tam, L. Choshen, C. A. Raffel, and M. Bansal, "Ties-merging: Resolving interference when merging models," in *NeurIPS*, vol. 36, LA, USA, 2023, pp. 7093–7115.

[32] L. Yu, B. Yu, H. Yu, F. Huang, and Y. Li, "Language models are super mario: Absorbing abilities from homologous models as a free lunch," in *ICML*, WIE, AT, 2024.

[33] C. Mavromatis, P. Karypis, and G. Karypis, "Pack of LLMs: Model fusion at test-time via perplexity optimization," in *COLM*, PA, USA, 2024.

[34] C. Goddard, S. Siriwardhana, M. Ehghaghi, L. Meyers, V. Karpukhin, B. Benedict, M. McQuade, and J. Solawetz, "Arcee's MergeKit: A toolkit for merging large language models," in *ACL*, FL, USA, 2024, pp. 477–485.

[35] D. Liu, Z. Wang, B. Wang, W. Chen, C. Li, Z. Tu, D. Chu, B. Li, and D. Sui, "Checkpoint merging via bayesian optimization in llm pretraining," 2024, arXiv:2403.19390.

[36] T. Fu, D. Cai, L. Liu, S. Shi, and R. Yan, "Disperse-then-merge: Pushing the limits of instruction tuning via alignment tax reduction," in *ACL*, BKK, TH, 2024, pp. 2967–2985.

[37] H. A. A. K. Hammoud, U. Michieli, F. Pizzati, P. Torr, A. Bibi, B. Ghanem, and M. Ozay, "Model merging and safety alignment: One bad model spoils the bunch," in *ACL*, FL, USA, 2024, pp. 13 033–13 046.

[38] Y. Zhu, R. Xia, and J. Zhang, "Dppa: Pruning method for large language model to model merging," 2024, arXiv:2403.02799.

[39] S. Kim, J. Suk, S. Longpre, B. Y. Lin, J. Shin, S. Welleck, G. Neubig, M. Lee, K. Lee, and M. Seo, "Prometheus 2: An open source language model specialized in evaluating other language models," in *ACL*, FL, USA, 2024, pp. 4334–4353.

[40] Y. Lin, H. Lin, W. Xiong, S. Diao, J. Liu, J. Zhang, R. Pan, H. Wang, W. Hu, H. Zhang, H. Dong, R. Pi, H. Zhao, N. Jiang, H. Ji, Y. Yao, and T. Zhang, "Mitigating the alignment tax of RLHF," in *ACL*, FL, USA, 2024, pp. 580–606.

[41] A. Tang, L. Shen, Y. Luo, L. Ding, H. Hu, B. Du, and D. Tao, "Concrete subspace learning based interference elimination for multi-task model fusion," 2023, arXiv:2312.06173.

[42] C. Fang, X. Li, Z. Fan, J. Xu, K. Nag, E. Korpeoglu, S. Kumar, and K. Achan, "Llm-ensemble: Optimal large language model ensemble method for e-commerce product attribute value extraction," in *ACM SIGIR*, NY, USA, 2024, p. 2910–2914.

[43] T. Akiba, M. Shing, Y. Tang, Q. Sun, and D. Ha, "Evolutionary optimization of model merging recipes," 2024, arXiv:2403.13187.

[44] F. Wan, Z. Yang, L. Zhong, X. Quan, X. Huang, and W. Bi, "Knowledge fusion of chat llms: A preliminary technical report," 2024, arXiv:2402.16107.

[45] H. Hoang, H. Khayrallah, and M. Junczys-Dowmunt, "On-the-fly fusion of large language models and machine translation," in *ACL*, CDMX, MX, 2024, pp. 520–532.

[46] Y.-C. Yu, C. C. Kuo, Y. Ziqi, C. Yucheng, and Y.-S. Li, "Breaking the ceiling of the LLM community by treating token generation as a classification for ensembling," in *ACL*, FL, USA, 2024, pp. 1826–1839.

[47] A. Ramé, N. Vieillard, L. Hussenot, R. Dadashi, G. Cideron, O. Bachem, and J. Ferret, "Warm: On the benefits of weight averaged reward models," in *ICML*, WIE, AT, 2024.

[48] Y. Zhai, H. Zhang, Y. Lei, Y. Yu, K. Xu, D. Feng, B. Ding, and H. Wang, "Uncertainty-penalized reinforcement learning from human feedback with diverse reward lora ensembles," 2023, arXiv:2401.00243.

[49] A. M. Ahmed, R. Rafailov, S. Sharkov, X. Li, and S. Koyejo, "Scalable ensembling for mitigating reward overoptimisation," in *ICLR*, WIE, AT, 2024.

[50] J. Eisenstein, C. Nagpal, A. Agarwal, A. Beirami, A. N. D'Amour, K. D. Dvijotham, A. Fisch, K. A. Heller, S. R. Pfohl, D. Ramachandran, P. Shaw, and J. Berant, "Helping or herding? reward model ensembles mitigate but do not eliminate reward hacking," in *COLM*, PA, USA, 2024.

[51] S. Zhang, Z. Chen, S. Chen, Y. Shen, Z. Sun, and C. Gan, "Improving reinforcement learning from human feedback with efficient reward model ensemble," 2024, arXiv:2401.16635.

[52] H. Wang, F. M. Polo, Y. Sun, S. Kundu, E. Xing, and M. Yurochkin, "Fusing models with complementary expertise," in *ICLR*, WIE, AT, 2024.

[53] D. Jiang, X. Ren, and B. Y. Lin, "LLM-blender: Ensembling large language models with pairwise ranking and generative fusion," in *ACL*, TO, CA, 2023, pp. 14 165–14 178.

[54] B. Lv, C. Tang, Y. Zhang, X. Liu, P. Luo, and Y. Yu, "URG: A unified ranking and generation method for ensembling language models," in *ACL*, BK, TH, 2024, pp. 4421–4434.

[55] C. Si, W. Shi, C. Zhao, L. Zettlemoyer, and J. L. Boyd-Graber, "Getting moRE out of mixture of language model reasoning experts," in *EMNLP*, SG, 2023.

[56] D. Oniani, J. Hilsman, H. Dong, F. Gao, S. Verma, and Y. Wang, "Large language models vote: Prompting for rare disease identification," 2024, arXiv:2308.12890.

[57] T. Shnitzer, A. Ou, M. Silva, K. Soule, Y. Sun, J. Solomon, N. Thompson, and M. Yurochkin, "LLM routing with benchmark datasets," in *NeurIPS*, LA, USA, 2023.

[58] K. Lu, H. Yuan, R. Lin, J. Lin, Z. Yuan, C. Zhou, and J. Zhou, "Routing to the expert: Efficient reward-guided ensemble of large language models," in *ACL*, CDMX, MX, 2024, pp. 1964–1974.

[59] D. Ding, A. Mallick, C. Wang, R. Sim, S. Mukherjee, V. Rühle, L. V. S. Lakshmanan, and A. H. Awadallah, "Hybrid LLM: Cost-efficient and quality-aware query routing," in *ICLR*, WIE, AT, 2024.

[60] K. A. Srivatsa, K. Maurya, and E. Kochmar, "Harnessing the power of multiple minds: Lessons learned from LLM routing," in *ACL*, CDMX, MX, 2024, pp. 124–134.

[61] A. Mohammadshahi, A. R. Shaikh, and M. Yazdani, "Routoo: Learning to route to large language models effectively," 2024, arXiv:2401.13979.

[62] L. Chen, M. Zaharia, and J. Zou, "Frugalgpt: How to use large language models while reducing cost and improving performance," 2023, arXiv:2305.05176.

[63] D. Dohan, W. Xu, A. Lewkowycz, J. Austin, D. Bieber, R. G. Lopes, Y. Wu, H. Michalewski, R. A. Saurous, J. Sohl-dickstein, K. Murphy, and C. Sutton, "Language model cascades," 2022, arXiv:2207.10342.

[64] S. Shen, L. Hou, Y. Zhou, N. Du, S. Longpre, J. Wei, H. W. Chung, B. Zoph, W. Fedus, X. Chen, T. Vu, Y. Wu, W. Chen, A. Webson, Y. Li, V. Y. Zhao, H. Yu, K. Keutzer, T. Darrell, and D. Zhou, "Mixture-of-experts meets instruction tuning: A winning combination for large language models," in *ICLR*, WIE, AT, 2024.

[65] M. Sakota, M. Peyrard, and R. West, "Fly-swat or cannon? cost-effective language model choice via meta-modeling," in *ACM WSDM*, vol. 35, MID, MX, 2024, p. 606–615.

[66] G. McDowell, *Cracking the Coding Interview: 189 Programming Questions and Solutions*. CareerCup, LLC, 2015. [Online]. Available: https://books.google.co.uk/books?id=jD8iswEACAAJ

[67] J. Lu, Z. Pang, M. Xiao, Y. Zhu, R. Xia, and J. Zhang, "Merge, ensemble, and cooperate! a survey on collaborative strategies in the era of large language models," 2024, arXiv:2407.06089.

[68] E. Yang, L. Shen, G. Guo, X. Wang, X. Cao, J. Zhang, and D. Tao, "Model merging in llms, mllms, and beyond: Methods, theories, applications and opportunities," 2024, arXiv:2408.07666.

[69] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, vol. 30, CA, USA, 2017.

[70] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *ACL*, MN, USA, 2019, pp. 4171–4186.

[71] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, A. Webson, S. S. Gu, Z. Dai, M. Suzgun, X. Chen, A. Chowdhery, A. Castro-Ros, M. Pellat, K. Robinson, D. Valter, S. Narang, G. Mishra, A. Yu, V. Zhao, Y. Huang, A. Dai, H. Yu, S. Petrov, E. H. Chi, J. Dean, J. Devlin, A. Roberts, D. Zhou, Q. V. Le, and J. Wei, "Scaling instruction-finetuned language models," *JMLR*, vol. 25, no. 70, pp. 1–53, 2024.

[72] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. Le, and R. Salakhutdinov, "Transformer-XL: Attentive language models beyond a fixed-length context," in *ACL*, FL, IT, 2019, pp. 2978–2988.

[73] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[74] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," 2019, arXiv:1904.10509.

[75] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," 2019. [Online]. Available: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[76] M. Rei, "Semi-supervised multitask learning for sequence labeling," in *ACL*, VAN, CA, 2017, pp. 2121–2130.

[77] M. E. Peters, W. Ammar, C. Bhagavatula, and R. Power, "Semi-supervised sequence tagging with bidirectional language models," in *ACL*, VAN, CA, 2017, pp. 1756–1765.

[78] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, arXiv:1707.06347.

[79] Z. Yang, D. Xue, S. Qian, W. Dong, and C. Xu, "Ldre: Llm-based divergent reasoning and ensemble for zero-shot composed image retrieval," in *ACM SIGIR*, NY, USA, 2024, p. 80–90.

[80] G. Barabucci, V. Shia, E. Chu, B. Harack, K. Laskowski, and N. Fu, "Combining multiple large language models improves diagnostic accuracy," *NEJM AI*, vol. 1, no. 11, 2024.

[81] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *NIPS*, MTL, CA, 2015.

[82] Y. Fu, H. Peng, L. Ou, A. Sabharwal, and T. Khot, "Specializing smaller language models towards multi-step reasoning," in *ICML*, vol. 202. HI, USA: PMLR, 2023, pp. 10 421–10 430.

[83] Y. Huang, X. Feng, B. Li, Y. Xiang, H. Wang, T. Liu, and B. Qin, "Ensemble learning for heterogeneous large language models with deep parallel collaboration," in *NeurIPS*, VAN, CA, 2024.

[84] Y. Xu, J. Lu, and J. Zhang, "Bridging the gap between different vocabularies for llm ensemble," in *NAACL*, CDMX, MX, 2024.

[85] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," in *ICLR*, WIE, AT, 2021.

[86] N. Shazeer, "Glu variants improve transformer," 2020, arXiv:2002.05202.

[87] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," 2016, arXiv:1606.06565.

[88] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[89] K. Brantley, W. Sun, and M. Henaff, "Disagreement-regularized imitation learning," in *ICLR*, AA, ET, 2020.

[90] M. Ravaut, S. Joty, and N. Chen, "SummaReranker: A multi-task mixture-of-experts re-ranking framework for abstractive summarization," in *ACL*, DUB, IRL, 2022, pp. 4504–4524.

[91] W. Yuan, G. Neubig, and P. Liu, "BARTScore: Evaluating generated text as text generation," in *NeurIPS*, 2021.

[92] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in *EMNLP-IJCNLP*. HK, CH: ACL, 2019, pp. 3982–3992.

[93] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," in *ACL*, DUB, IRL, 2022, pp. 8086–8098.

[94] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *NeurIPS*, LA, USA, 2022.

[95] W. Chen, X. Ma, X. Wang, and W. W. Cohen, "Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks," *TMLR*, 2023.

[96] C. Si, Z. Gan, Z. Yang, S. Wang, J. Wang, J. L. Boyd-Graber, and L. Wang, "Prompting GPT-3 to be reliable," in *ICLR*, KG, RW, 2023.

[97] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. V. Le, and E. H. Chi, "Least-to-most prompting enables complex reasoning in large language models," in *ICLR*, KG, RW, 2023.

[98] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *IEEE Neural Computation*, vol. 3, no. 1, pp. 79–87, 1991.

[99] T. Gale, D. Narayanan, C. Young, and M. Zaharia, "Megablocks: Efficient sparse training with mixture-of-experts," in *MLSys*, vol. 5. Curan, 2023, pp. 288–304.

[100] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "BERTScore: Evaluating text generation with bert," in *ICLR*, AA, ET, 2020.

[101] P. Pobrotyn, T. Bartczak, M. Synowiec, R. Białobrzeski, and J. Bojar, "Context-aware learning to rank with self-attention," in *SIGIR eCom*, 2020.

[102] R. Puri, D. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. T. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, "CodeNet: A large-scale ai for code dataset for learning a diversity of coding tasks," in *NeurIPS*, vol. 1, 2021.

[103] S. Gundabathula and S. Kolar, "PromptMind team at MEDIQA-CORR 2024: Improving clinical text correction with error categorization and LLM ensembles," in *ACL ClinicalNLP*, CDMX, MX, 2024, pp. 367–373.

[104] G. Barabucci, V. Shia, E. Chu, B. Harack, K. Laskowski, and N. Fu, "Combining multiple large language models improves diagnostic accuracy," *NEJM AI*, vol. 1, no. 11, p. AIcs2400502, 2024.

[105] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *ICLR*, KG, RW, 2023.

[106] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "CodeBERTScore: Evaluating code generation with pretrained models of code," in *ACL EMNLP*, SG, 2023, pp. 13 921–13 937.

[107] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: a method for automatic evaluation of machine translation,” in *ACL*, PA, USA, 2002, p. 311–318.

[108] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “CodeBLEU: a method for automatic evaluation of code synthesis,” 2020, arXiv:2009.10297.