

# Wisdom and Delusion of LLM Ensembles for Code Generation and Repair

Fernando Vallecillos-Ruiz  
fernando@simula.no  
Simula Research Laboratory  
Oslo, Norway

Max Hort  
maxh@simula.no  
Simula Research Laboratory  
Oslo, Norway

Leon Moonen  
leon.moonen@computer.org  
Simula Research Laboratory  
Oslo, Norway

## Abstract

Today’s pursuit of a single Large Language Model (LLM) for all software engineering tasks is resource-intensive and overlooks the potential benefits of complementarity, where different models contribute unique strengths. However, the degree to which coding LLMs complement each other and the best strategy for maximizing an ensemble’s potential are unclear, leaving practitioners without a clear path to move beyond single-model systems.

To address this gap, we empirically compare ten individual LLMs from five families, and three ensembles of these LLMs across three software engineering benchmarks covering code generation and program repair. We assess the complementarity between models and the performance gap between the best individual model and the ensembles. Next, we evaluate various selection heuristics to identify correct solutions from an ensemble’s candidate pool.

We find that the theoretical upperbound for an ensemble’s performance can be 83% above the best single model. Our results show that consensus-based strategies for selecting solutions fall into a “popularity trap,” amplifying common but incorrect outputs. In contrast, a diversity-based strategy realizes up to 95% of this theoretical potential, and proves effective even in small two-model ensembles, enabling a cost-efficient way to enhance performance by leveraging multiple LLMs.

## CCS Concepts

• **Computing methodologies** → **Ensemble methods**; Natural language processing; • **Software and its engineering** → *Software development techniques*; • **General and reference** → Empirical studies.

## Keywords

Code Generation, Large Language Models, Automatic Program Repair, Ensemble Learning

## 1 Introduction

Large Language Models (LLMs) have demonstrated remarkable skill in software engineering tasks, from code generation to Automatic Program Repair (APR) [6, 12]. Their ability to generate human-like code has promised to enhance developer productivity and accelerate software development [25]. This has resulted in an arms race to investigate and develop a single, superior model that can outperform all others across any benchmark [5, 37], aiming to find a definite answer to a question that has become more and more common: “Which LLM is the best for coding-related tasks?”

The core problem with this approach is the implicit assumption that a single model can eventually dominate all others across the heterogeneous landscape of programming problems [3, 32].

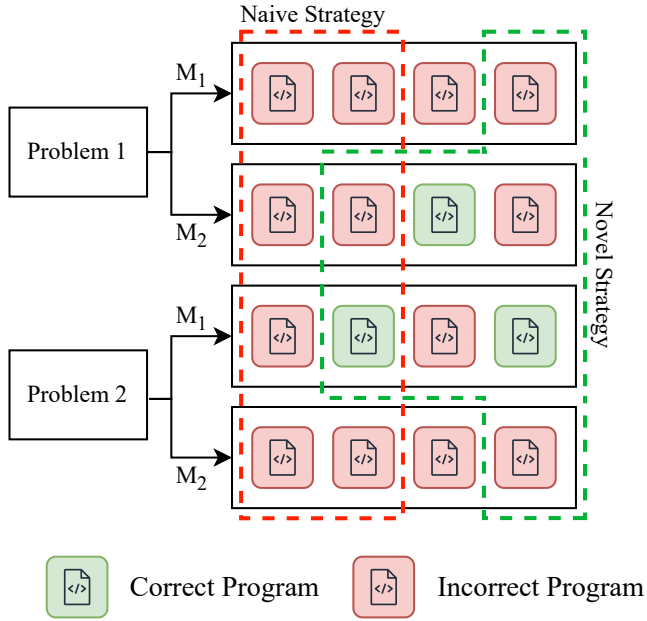
The search for a single best model overlooks experience in related fields, such as in the pre-LLM era of APR. It was well accepted that the landscape of bugs was too diverse for any single repair tool to tackle [45]. State-of-the-art results were achieved by creating ensembles of diverse techniques or tools (e.g. combining search-based, constraint-based, and template-based approaches) where the collective result exceeded that of any individual component. We argue that this principle of complementarity is still fundamental to modern LLMs, where no single model can solve all problems. Models trained on diverse datasets with varying architectures or numbers of parameters likely develop unique problem-solving capabilities. However, the field currently lacks a clear understanding of the degree to which each model solves a unique set of problems that other models cannot. Recently, the need for fundamental understanding has been rising due to the increased interest in multi-agent systems where multiple LLM-based agents collaborate on complex software engineering tasks [10]. The rationale behind multi-agent systems and ensembles of models is that the combination of multiple models can overcome their individual limitations and outperform any single model [22].

Establishing model complementarity is necessary, but not the only requirement to obtain practical benefits. Each model in an ensemble generates multiple candidates, creating a noisy pool of potential solutions. Naive selection from this pool can be ineffective. Intuitive heuristics, such as selecting the most common solution might seem promising, but could lead models to converge on plausible but incorrect answers. Alternatively, one could leverage confidence scores of each model, but the effectiveness of these signals coming from models with different architectures remains an open question. Without a systematic evaluation, the practical benefits of model complementarity will remain theoretical, leaving practitioners without a reliable method to realize this potential. Figure 1 shows that while model complementarity can create a pool of candidates containing solutions for two problems, only an effective selection heuristic can realize that potential.

Therefore, we perform an empirical study to systematically explore and quantify the potential of LLM ensembles for software engineering tasks. Our goal is not to compete for a new state-of-the-art score on a benchmark, but rather to provide fundamental understanding on ensembles and how they compare to a single-model paradigm. To achieve this, we select a set of ten instruction-tuned LLMs with strong coding capabilities, spanning five different model families. This set consists of five smaller models (~7B parameters) and five larger models (~14B parameters). We evaluate these models



This work is licensed under a Creative Commons Attribution 4.0 International License.



**Figure 1: Conceptual overview of the selection process from an LLM ensemble’s output pool where only 4 programs can be selected per problem.**

across three common benchmarks covering two software engineering tasks: HumanEval-Java [15] and Defects4J [16] for APR, and LiveCodeBench [13] covering code generation.

Our investigation consists of two phases. First, we establish a theoretical foundation for ensembles by analyzing model complementarity among models in the same family and those with different parameter counts. This allows us to quantify the performance gap between an individual model and the theoretical potential of an ensemble. Secondly, having established this performance ceiling, we propose and evaluate a set of heuristics to select candidates from the aggregated output pool as illustrated in Figure 1. Through this analysis, we aim to provide concrete, empirically driven strategies to effectively realize the potential of multiple LLMs.

Our contributions are as follows:

- We conduct an extensive empirical study with ten LLMs from five families across three code-related benchmarks, quantifying and analyzing for the first time the performance gap between single models and ensembles for software engineering tasks.
- We provide empirical evidence that LLMs exhibit significant complementarity where even smaller models contribute unique solutions not found by large state-of-the-art models.
- We quantify the performance ceiling offered by ensembles of models, demonstrating a potential performance increase of up to 83% in the number of problems solved by the best individual model.
- We systematically evaluate various selection heuristics to identify correct solutions from a pool of candidates.

- We identify the challenge of the “popularity trap” where models frequently converge on the same incorrect solution and provide a strategy to counter this trap that realizes up to 95% of the ensemble’s theoretical potential.
- After initial experiments with ensembles of five or ten models, we extend our analysis to ensembles of two models to confirm our findings in resource-constrained environments.
- We release a replication package containing our experimental framework, raw data, and analysis scripts to encourage further research into LLM ensembles.<sup>11</sup>

## 2 Related Work

Lu et al. [20] conducted a survey on collaborative strategies for LLMs. In total, they devised three collaboration types: merging, cooperation, and ensemble. The *merging* of LLMs is concerned with their parameters. As such, the parameters of multiple LLMs are fused into a single model. *Cooperation* encompasses more complex types of interactions, such as the use of LLMs to check the factuality of outputs generated by another or the use of small LLMs to compress inputs for larger ones with the goal of improving efficiency. *Ensemble* methods combine outputs generated by multiple LLMs. This can occur before, during, or after inference. Specifically, we are interested in LLM ensembles after inference, in which collaboration is achieved by selecting a subset of outputs generated from all LLMs.

In addition to the survey by Lu et al. [20], Ashiga et al. [1] surveyed ensemble methods for text and code generation with LLMs. They devised seven types of LLM ensembles: weight merging, knowledge fusion, mixture-of-experts, reward ensemble, output ensemble, routing, and cascading. Among these, the most popular approach for using ensembles was the use of mixture-of-experts [21], where the output of a single LLM is determined from an architecture perspective. Similar to the mixture-of-agents approach, Xue et al. [41] created ensembles with code generated in multiple languages from the same model (similar to MoE). Other code-related tasks addressed with ensembles include vulnerability detection [11, 28, 35], code search [4] and code generation [2]. Here, Chen et al. [2] employed two LLMs to iteratively generate code, and test cases for the respective other LLM.

The approach LLM-BLENDER, proposed by Jiang et al. [14], used ensembles of LLMs for instruction-based tasks and is able to consistently outperform single models. LLM-BLENDER consists of two components: PAIRRANKER and GENFUSER. PAIRRANKER ranks outputs by performing a pairwise comparison between every generated output, to determine the better of the two candidates. For this purpose, metrics such as BERTScore [44], BLEURT [31], and BARTScore [43] have been used to score each output according to its similarity to the ground truth. An alternative is the ranking of output pairs with ChatGPT, which did not require access to ground truth information. Unlike our goal to rank and choose multiple outputs from an ensemble of LLMs, PAIRRANKER ranks outputs, and GENFUSER uses this information to generate a single, final response.

Other output ranking approaches, which can support the selection of outputs when dealing with an ensemble of LLMs, include the computation of cosine similarity between input and output [19] or

masked language modeling to compute the likelihood of generated outputs [30]. Ravaut et al. [26] proposed SummaReranker, which learned a ranking model to determine the best solution from a set of candidates. For the task of program repair, confidence-based ranking is popular (i.e. how confident are the LLMs that the generated output solves the problem). As such, entropy has been used to rank the quality of patches [38, 39, 42].

While several approaches have been proposed for enabling the collaboration of LLMs, we notice a lack of research on outputs ensembles (i.e. selecting multiple outputs generated from an ensemble of LLMs), in particular when dealing with software engineering tasks. Therefore, we set out to perform an empirical evaluation of output ensembles for two software engineering tasks (i.e. code generation and program repair).

### 3 Experiment Design

#### 3.1 Research Questions

We aim to answer the following research questions in our work:

- RQ1 What are the performance differences between individual models vs. an ensemble of models for software engineering tasks?
  - RQ1.1 To what degree do LLMs exhibit complementarity by solving unique sets of problems?
  - RQ1.2 How large is the performance gap between an ensemble’s theoretical maximum and the best-performing individual model?

To answer RQ1, we employ 10 different LLMs and 3 ensembles (Section 3.2). We evaluate the performance of each individual model and each ensemble configuration on three benchmarks (Section 3.3). We first analyze model complementarity by studying the problem-solving capabilities within and across LLM families. For each ensemble, we calculate the theoretical maximum score, which indicates the total number of problems solved by at least one model in a specific ensemble. This quantifies the performance gap between a single model and the ensemble’s theoretical maximum, establishing the ceiling for an ideal selection strategy.

- RQ2 What heuristics are the most effective at achieving the theoretical potential of an LLM ensemble?
  - RQ2.1 How do selection metrics and strategies compare in their ability to identify correct solutions from the pool of candidates?
  - RQ2.2 How consistent are the selection strategies when reducing the ensemble size to two models?

To address RQ2, we first implement and evaluate different heuristics to select outputs generated by ensembles of 5 or 10 models (Section 3.2). Each heuristic is composed of a selection metric (Section 3.6) and a selection strategy (Section 3.8). We employ selection metrics that assign scores to each output based on either the confidence of the models or the similarity between outputs. We then implement three selection strategies to favor consensus, disagreement, or diversity. This allows us to compare each heuristic against the theoretical best possible score established in RQ1. We then investigate the performance of these strategies on smaller ensembles composed of only 2 models. We evaluate if the trends appearing on bigger ensembles persist and compare their performance against

**Table 1: Overview of language models used.**

| Family    | Model Name                         | Size | Abbr.  |
|-----------|------------------------------------|------|--------|
| CodeLlama | CodeLlama-7b <sup>1</sup>          | 7B   | $CL_S$ |
|           | CodeLlama-13b <sup>2</sup>         | 13B  | $CL_L$ |
| DeepSeek  | DeepSeekCoder-6.7b <sup>3</sup>    | 6.7B | $DS_S$ |
|           | DeepSeekCoder-V2-Lite <sup>4</sup> | 16B  | $DS_L$ |
| Gemma     | CodeGemma-7b <sup>5</sup>          | 7B   | $GM_S$ |
|           | Gemma-3-12b <sup>6</sup>           | 12B  | $GM_L$ |
| Mistral   | Ministral-8B <sup>7</sup>          | 8B   | $MI_S$ |
|           | Mistral-Nemo <sup>8</sup>          | 12B  | $MI_L$ |
| Qwen3     | Qwen3-8B <sup>9</sup>              | 8B   | $QW_S$ |
|           | Qwen3-14B <sup>10</sup>            | 14B  | $QW_L$ |

a Naive baseline, thereby assessing their utility in more resource-constrained environments.

#### 3.2 Models and Ensembles

To evaluate our research questions, we select a diverse set of 10 publicly available, instruction-tuned LLMs from five model families, all with coding capabilities. For each family, we select two models of different parameter sizes. For our analysis, we group these models into two categories: Small Models, which comprises models with approximately 7 to 8 billion parameters, and Large Models, which includes models in the 12 to 16 billion parameter range. Table 1 provides an overview of the models in our study, along with abbreviations used for conciseness in Tables and Figures throughout the paper. The abbreviations contain a subscript indicating if the model is the smaller variant (e.g.  $MI_S$ ) or the larger variant (e.g.  $MI_L$ ) in the family.

Based on these 10 models, we have created three ensembles. The ensemble  $Ens_{all}$  contains all ten models. The ensemble  $Ens_S$  contains the five small variants for each family. Finally, the ensemble  $Ens_L$  contains the five large variants for each family.

#### 3.3 Benchmarks

To ensure a comprehensive evaluation, we select three widely-used benchmarks representing two software engineering tasks: Automatic Program Repair (APR) and code generation. For the APR task, we use HumanEval-Java and Defects4J. HumanEval-Java [15] is a Java-based variant of the code synthesis benchmark. This benchmark consists of 164 single-function bugs. Defects4J v2.0 [16] comprises 835 bugs from large, open-source Java projects. We follow

<sup>1</sup><https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf>

<sup>2</sup><https://huggingface.co/codellama/CodeLlama-13b-Instruct-hf>

<sup>3</sup><https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct>

<sup>4</sup><https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct>

<sup>5</sup><https://huggingface.co/google/codegemma-7b-it>

<sup>6</sup><https://huggingface.co/google/gemma-3-12b-it>

<sup>7</sup><https://huggingface.co/mistralai/Ministral-8B-Instruct-2410>

<sup>8</sup><https://huggingface.co/mistralai/Mistral-Nemo-Instruct-2407>

<sup>9</sup><https://huggingface.co/Qwen/Qwen3-8B>

<sup>10</sup><https://huggingface.co/Qwen/Qwen3-14B>

the classification of previous work [40] and select the 525 single-function bugs. Both APR benchmarks provide a potential fix along with unit tests to evaluate the plausibility of generated patches.

For the code generation task, we consider LiveCodeBench [13], a recent benchmark consisting of 1180 problems from competitive programming platforms compiled from May 2023 to May 2025. From these, we select a subset of the latest 454 problems (from August 2024) set as default by the authors on their leaderboard. This selection attempts to limit the data leakage on newer models. Each problem consists of a problem description and may contain some starter code. Unlike the previous benchmarks, LiveCodeBench does not contain any potential solution and relies only on a set of tests to assess plausibility. By combining code-generation and program repair benchmarks, we aim to study the dynamics of LLM ensembles on different software engineering tasks and not constrain our analysis to a single one.

### 3.4 Implementation

The pipeline begins by building the initial prompt tailored to each benchmark. We use a beam-based search decoding strategy with no stochastic sampling to keep all outputs deterministic and reproducible. For the two APR benchmarks, the input consists of a function containing a bug. The bug is delimited using the `<bug_start>` and `<bug_end>`. The following template is used to generate the prompt:

```
"""
The input is buggy code. The bug starts from
'<bug_start>' and ends at '<bug_end>'.
Please fix the following code delimited by
triple backticks, remove the '<bug_start>'
and '<bug_end>' markers, and return the
complete method fixed wrapped in triple backticks.
```java
{buggy_function}
```
"""
```

On the other hand, the prompt for the code generation benchmark starts with a description of a problem which may include starting code. We follow the template proposed in the original work [13]:

```
"""
You are an expert Python programmer. You will be given a
question (problem specification) and will generate a
correct Python program that matches the specification
and passes all tests. You will NOT return anything
except for the program.

### Question: {question_content}

### Format: {formatting_message}

```python
{starter_code}
# YOUR CODE HERE
```
"""
```

The formatting message for questions without starter code is:

Read the inputs from stdin solve the problem and write the answer to stdout (do not directly test on the sample inputs). Enclose your code within delimiters as follows.

The formatting message for questions with started code is:

You will use the following starter code to write the solution to the problem and enclose your code within delimiters.

The outputs generated in either of the benchmarks are then parsed by extracting the blocks of text delimited by the triple back-quote symbol (```) that can be followed by a keyword (``java` or ``python`). Since models may output partial answers before or after the complete solution for APR problems, if multiple blocks are delimited, we select the one that contains a full method declaration.

In the validation phase, we execute the code extracted (after being inserted in the appropriate context) and run the related tests. The tests determine if the code is plausible or not.

### 3.5 Family Advantage Index

We introduce the Family Advantage Index ( $FAI_z$ ). This z-score quantifies how much more likely a large model is to solve a hard problem than its smaller counterpart solved, relative to the average performance of other large models. We define a hard problem as one not solved by all small models, ensuring that very easy problems solved by everyone are excluded from the calculation. A positive  $FAI_z$  score indicates the degree of family advantage, i.e. the larger model is more likely to solve the same hard problems than its smaller counterpart solved compared to other larger models. However, it is critical to acknowledge potential limitations of this score.  $FAI_z$  is sensitive to performance variations of other models used to calculate the average, and to the definition of what is considered a hard problem.

Formally, let  $\mathcal{H}$  be the set of hard problems (those solved by at least one but not all small models in our study). Given a pair of models ( $M_S, M_L$ ) from the same family, we calculate the conditional probability that  $M_L$  solves a hard problem, given  $M_S$  solved it:  $p_L = P(M_L | M_S, \mathcal{H})$ . We then compute the mean ( $\mu$ ) and standard deviation ( $\sigma$ ) of this same conditional probability for all other large models in our set. The Family Advantage Index is the resulting z-score:

$$FAI_z(M_S \rightarrow M_L) = \frac{p_L - \mu}{\sigma} \quad (1)$$

### 3.6 Selection Metrics

The metrics chosen can be divided into two categories: output-based metrics and confidence-based metrics. **Output-based** metrics measure semantic and syntactic properties between two pieces of code. We select two output-based metrics: CodeBERTScore [46] and CodeBLEU [27]. CodeBERTScore uses the embeddings from the CodeBERT model [7] to compute the cosine similarity between two pieces of code. We choose CodeBERTScore F3 since it is recommended by the authors for functional correctness. CodeBLEU measures syntactic similarity by adapting BLEU [24] to source code. It calculates a weighted combination of the original BLEU score, n-gram match, abstract syntax tree match, and data-flow match. For

both metrics, we compute the final score of a candidate by summing over all scores of said candidate against the rest of the candidates for the same problem.

On the other hand, **confidence-based** metrics employ a model’s internal confidence to assess an output. Prior work [38, 42] has successfully used confidence-based metrics to rank patches. We select three confidence-based metrics: NLL / Byte, Entropy / Byte, and Normalized Sum Entropy.

Let  $t_{1:n}$  be the tokens of a patch, and let  $p_i$  denote the model probability assigned to token  $t_i$  given the preceding tokens. We define negative log-likelihood (NLL) and the per-position predictive entropy as:

$$\text{NLL} = -\sum_{i=1}^n \log p_i, \quad E_i = -\sum_{v \in \mathcal{V}} q_i(v) \log q_i(v), \quad (2)$$

where  $q_i(\cdot)$  is the model’s distribution at position  $i$ ,  $\mathcal{V}$  is the vocabulary, and all logarithms are natural (in nats). Let  $m$  be the number of bytes (in UTF-8) in the generated patch, and let  $\log |\mathcal{V}|$  be the maximum entropy of a uniform distribution over the vocabulary. We then compute:

$$\text{nll\_per\_byte} = \frac{\text{NLL}}{m}, \quad (3)$$

$$\text{entropy\_per\_byte} = \frac{\sum_{i=1}^n E_i}{m}, \quad (4)$$

$$\text{sum\_entropy\_norm} = \frac{\sum_{i=1}^n E_i}{\log |\mathcal{V}|}. \quad (5)$$

The first two metrics are normalized by bytes to mitigate tokenization effects and enable comparison across models and tokenizers. This type of normalization cannot be applied to the entropy sum. However, we still employ this metric inspired by Xia et al. [38] who obtained better outcomes favoring shorter and simpler code following Occam’s razor hypothesis [34]. To reduce the effects of vocabulary differences between the models on the entropy sum, we normalize according to the vocabulary size of each tokenizer. For these 3 metrics, we calculate the final score of a candidate by summing over all confidence-based scores of said candidate given by all the models in the ensemble.

### 3.7 Candidate Pool Construction

For every problem in our benchmarks, we prompt each of the 10 models to generate  $n = 10$  outputs. The outputs from all models are aggregated in a large pool of candidates. From this set, we select a subset of  $k = 10$  outputs. This constraint is motivated by two practical considerations. First, developers are unlikely to review more than 10 patches [23]. Second, the computational cost of validating outputs can be high, especially for large-scale projects found in benchmarks such as Defects4J [18]. These decisions align with common practices balancing efficient use of resources with the need for a diverse candidate pool, and allow for comparison with previous work [15, 29, 33].

### 3.8 Selection Strategies

Given a set of scored candidates, we investigate three strategies to choose the final subset of  $k = 10$  candidates: Highest, Lowest,

**Table 2: [RQ1.1] Problems solved ( $M_S \rightarrow M_L$ ) and corresponding  $\text{FAI}_z$  per family and benchmark. Each cell shows the number of problems solved for the smaller model  $M_S$  and larger model  $M_L$ ; parentheses contain  $\text{FAI}_z$  computed on the benchmark’s set of hard problems.**

| Family    | HumanEval-Java                    | Defects4J                        | LiveCodeBench                     |
|-----------|-----------------------------------|----------------------------------|-----------------------------------|
| CodeLlama | 59→64 <sup>(−6.35)</sup>          | 42→47 <sup>(+0.00)</sup>         | 56→55 <sup>(−8.49)</sup>          |
| DeepSeek  | <b>110→122</b> <sup>(+1.02)</sup> | <b>89→112</b> <sup>(+1.11)</sup> | 87→110 <sup>(+0.04)</sup>         |
| Gemma     | 101→108 <sup>(+0.56)</sup>        | 84→87 <sup>(−0.46)</sup>         | 80→122 <sup>(+1.31)</sup>         |
| Mistral   | 108→104 <sup>(+0.21)</sup>        | 77→81 <sup>(+0.40)</sup>         | 87→92 <sup>(+0.02)</sup>          |
| Qwen      | 98→120 <sup>(+0.88)</sup>         | 65→111 <sup>(+1.90)</sup>        | <b>130→155</b> <sup>(+2.20)</sup> |

and Diversity. The first two strategies select the 10 outputs with the highest or lowest scores. The goal of these strategies is to select patches based on consensus and disagreement. The last strategy aims to maximize diversity within the set of candidate outputs. We employ a greedy approach starting with the two candidates with the highest and lowest scores. It then iteratively builds a set of 10 candidates by selecting candidates that maximize the distance to candidates already selected. The goal is to explore a wider range of potential solutions rather than focusing on minor variations of a single popular but incorrect candidate.

To establish a baseline, we also employ a Naive selection strategy that does not take into account any selection metric. This Naive strategy allows each model to contribute equally. Since we restrict the number of outputs per ensemble to  $k = 10$ , we build the selection with the first 1 or 2 outputs per model in ensembles of 10 or 5 models respectively.

### 3.9 Evaluation Metric

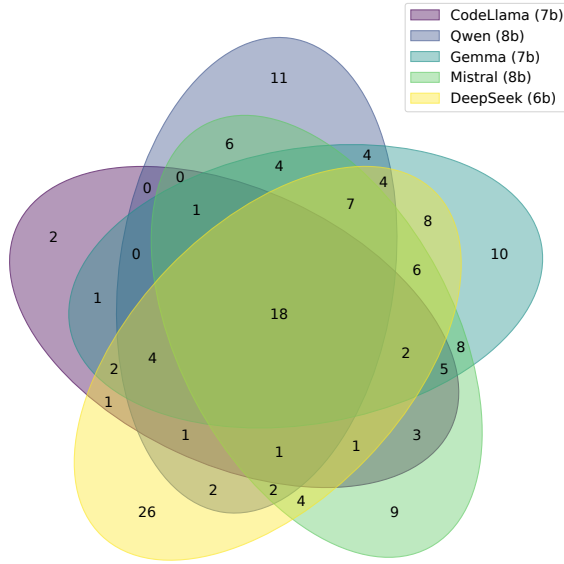
We assess the effectiveness of the different ensembles and strategies by measuring the number of problems in each benchmark with at least one plausible candidate. A candidate solution is considered plausible if it compiles successfully and passes the test suite associated with the problem. The candidate is considered implausible otherwise. Given that our experiments use 10 different models generating 10 outputs per problem, the test suite associated with each problem allows us to efficiently evaluate tens of thousands of candidate solutions.

## 4 Results and Discussion

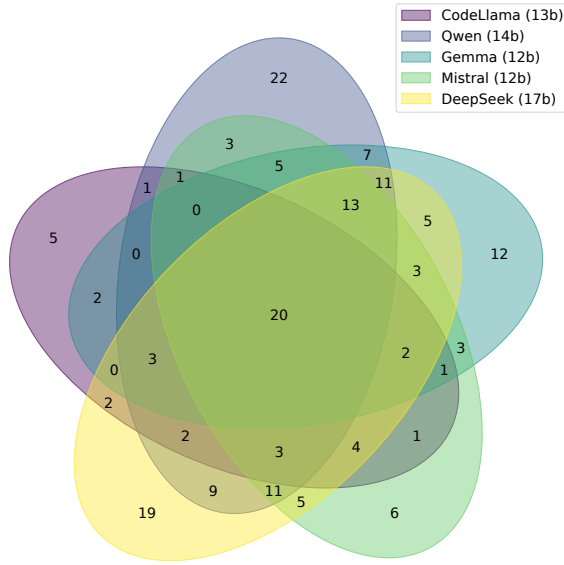
### 4.1 Results of RQ1

**4.1.1 Complementarity of LLMs.** An ensemble is expected to achieve better problem-solving capabilities than its best individual member. Thus, we need to analyze the ensemble’s theoretical performance ceiling before constructing it. To this end, we start by studying the relationship between the small ( $M_S$ ) and large ( $M_L$ ) variants of the same family before diving into bigger ensembles.

A common assumption is that larger models solve at least all the problems their smaller counterparts can [9, 17]. Our results quickly prove this assumption incorrect, as shown by the number of problems solved for each model in Table 2. For instance, CodeLlama (13b) solves fewer problems than CodeLlama (7b) on



(a) Small Models



(b) Large Models

**Figure 2: [RQ1.1] Venn diagram of problems solved for Defects4J benchmark for two sets of models indicating their complementarity.**

LiveCodeBench, proving it cannot be a superset. Another instance occurs in the Mistral family, where Mistral (8b) solves more problems on HumanEval-Java than Mistral (12b). This demonstrates that smaller models are still relevant and are not made redundant by larger versions.

**Table 3: [RQ1.2] Comparison of the number of problems solved by the single best small ( $M_S$ ) and large ( $M_L$ ) model against the theoretical maximum number of problems solved by an ensemble of five small models, five large models, and all ten models combined.**

| Benchmark      | Best single model (#)<br>( $M_S \rightarrow M_L$ ) | Theoretical max (#) |         |             |
|----------------|----------------------------------------------------|---------------------|---------|-------------|
|                |                                                    | $Ens_S$             | $Ens_L$ | $Ens_{all}$ |
| HumanEval-Java | 110→122                                            | 132                 | 139     | 141         |
| Defects4J      | 89→112                                             | 153                 | 181     | 205         |
| LiveCodeBench  | 130→155                                            | 154                 | 177     | 185         |

Despite this complementarity, the results also indicate that larger models tend to solve more problems overall. Therefore, we first study these intra-family relationships quantitatively.

To this end, we use the Family Advantage Index ( $FAI_z$ ) described in Section 3.5, a z-score that measures how many standard deviations a large model’s success rate on hard problems deviates from the average success rate of other large models on the same problems.

The number of problems solved for each model and the  $FAI_z$  for each family are shown in Table 2. Our results show that a family advantage often exists, but also exhibits high variance. The Qwen family exhibits a strong positive  $FAI_z$  on Defects4J (+1.90) and LiveCodeBench (+2.20). In contrast, the CodeLlama family shows an even stronger negative  $FAI_z$  on HumanEval-Java (−6.35) and LiveCodeBench (−8.49). These results indicate that even within the same family, individual model variants may offer distinct solutions. However, to increase the likelihood that an ensemble has the best chances to achieve higher outputs, including models from different families is preferred.

To confirm the viability of large-scale ensembles, we analyze the unique contribution of individual models to an ensemble. The existence of problems solved by only a single model, including models with a lower overall score, indicates complementarity between the models. We illustrate this complementarity through Venn diagrams composed of models of similar sizes. For brevity, we include only two out of nine diagrams, the remaining ones can be found in our public repository.<sup>11</sup> Figure 2a include smaller models while Figure 2b encompass their larger counterparts applied to Defects4J.

Even among the smaller models, the set of unique problems solved is substantial. DeepSeek (6B), the highest score in this benchmark, solves 26 unique bugs that no other small model could solve. Gemma (7b) solves only 5 problems fewer than DeepSeek (6b), yet contributes solutions for 10 unique bugs. While a model like CodeLlama (7b) solves fewer than half the number of problems of DeepSeek (6B), it is still able to provide solutions to two unique problems.

This trend repeats itself in the case of larger models. Lower-performing models like CodeLlama (13b) provide 5 unique solutions despite maintaining an overall score lower than half of the best model. In this case, the model with the most unique solution is Qwen (14b) with 22 unique bugs solved, followed by DeepSeek (16b) with 19 unique bugs solved.

Table 4: [RQ2.1] Problems solved HumanEval-Java. Bolding shows the best option per ensemble.

| Ensemble                 | CodeBERT F3 |     |            | CodeBLEU |     |     | NLL/Byte |     |            | Entropy/Byte |     |     | Sum Entropy |    |     | Naive | Best |
|--------------------------|-------------|-----|------------|----------|-----|-----|----------|-----|------------|--------------|-----|-----|-------------|----|-----|-------|------|
|                          | H           | L   | D          | H        | L   | D   | H        | L   | D          | H            | L   | D   | H           | L  | D   |       |      |
| <i>Ens<sub>S</sub></i>   | 64          | 108 | <b>129</b> | 64       | 110 | 128 | 75       | 109 | 125        | 75           | 111 | 125 | 103         | 90 | 125 | 112   | 132  |
| <i>Ens<sub>L</sub></i>   | 83          | 118 | 130        | 82       | 124 | 131 | 94       | 122 | <b>135</b> | 95           | 123 | 134 | 117         | 94 | 128 | 127   | 139  |
| <i>Ens<sub>all</sub></i> | 69          | 111 | 130        | 69       | 109 | 127 | 79       | 114 | <b>133</b> | 78           | 113 | 132 | 110         | 91 | 130 | 123   | 141  |

H / L / D = Highest / Lowest / Diversity

Table 5: [RQ2.1] Problems solved Defects4J. Bolding shows the best option per ensemble.

| Ensemble                 | CodeBERT F3 |     |     | CodeBLEU |     |            | NLL/Byte |     |     | Entropy/Byte |     |            | Sum Entropy |    |            | Naive | Best |
|--------------------------|-------------|-----|-----|----------|-----|------------|----------|-----|-----|--------------|-----|------------|-------------|----|------------|-------|------|
|                          | H           | L   | D   | H        | L   | D          | H        | L   | D   | H            | L   | D          | H           | L  | D          |       |      |
| <i>Ens<sub>S</sub></i>   | 26          | 114 | 134 | 22       | 118 | 135        | 77       | 80  | 133 | 75           | 93  | <b>137</b> | 100         | 54 | 128        | 88    | 153  |
| <i>Ens<sub>L</sub></i>   | 40          | 134 | 153 | 35       | 137 | <b>162</b> | 86       | 115 | 152 | 81           | 123 | 154        | 135         | 53 | 157        | 116   | 181  |
| <i>Ens<sub>all</sub></i> | 22          | 125 | 157 | 25       | 129 | 164        | 87       | 116 | 152 | 86           | 122 | 161        | 133         | 53 | <b>167</b> | 97    | 205  |

H / L / D = Highest / Lowest / Diversity

Table 6: [RQ2.1] Problems solved LiveCodeBench. Bolding shows the best option per ensemble.

| Ensemble                 | CodeBERT F3 |     |            | CodeBLEU |     |            | NLL/Byte |     |     | Entropy/Byte |     |     | Sum Entropy |     |            | Naive | Best |
|--------------------------|-------------|-----|------------|----------|-----|------------|----------|-----|-----|--------------|-----|-----|-------------|-----|------------|-------|------|
|                          | H           | L   | D          | H        | L   | D          | H        | L   | D   | H            | L   | D   | H           | L   | D          |       |      |
| <i>Ens<sub>S</sub></i>   | 102         | 103 | <b>143</b> | 102      | 101 | 141        | 99       | 106 | 140 | 99           | 105 | 138 | 103         | 107 | 138        | 142   | 154  |
| <i>Ens<sub>L</sub></i>   | 110         | 130 | 167        | 103      | 138 | <b>170</b> | 91       | 126 | 167 | 91           | 125 | 167 | 108         | 130 | 168        | 168   | 177  |
| <i>Ens<sub>all</sub></i> | 104         | 110 | 162        | 103      | 107 | 160        | 89       | 116 | 158 | 89           | 112 | 159 | 96          | 125 | <b>170</b> | 169   | 185  |

H / L / D = Highest / Lowest / Diversity

4.1.2 *Performance gap between ensembles and individual models.* The complementarity shown in the previous experiments indicates the existence of a performance ceiling. We define the theoretical maximum as the total number of unique problems solved by at least one model within a set of models, representing the best possible outcome under a perfect selection approach. The performance gap between the best single model of an ensemble to this theoretical maximum is substantial.

Table 3 presents this analysis across our three benchmarks. The greatest gap in performance occurs on the Defects4J benchmark. While the best small model solves 89 problems, an ensemble of small models could reach 153 fixed problems. Similarly, the best large model solves 112 problems, while an ensemble of large models can solve 181 problems. Comparing against the ensemble of all the models, the results indicate a potential performance improvement of 83% over the best individual model.

While the gap is largest for the Defects4J benchmark, it remains substantial across all benchmarks. Both benchmarks, LiveCodeBench and HumanEval-Java, also see an improvement between 10% to 20% between the highest score in an ensemble of models to the theoretical maximum score achieved by the ensemble. Furthermore, the data show that an ensemble of all models consistently outperforms ensembles of large and small models. For example, on Defects4J, the larger model ensemble could solve 181 problems, but adding the five smaller models increases this number by 24 unique solutions (a 13% increase). This finding further validates our earlier observations through the  $FAI_z$  analysis, confirming that smaller

models do not always produce subsets of solutions of their larger counterparts.

Our previous experiments have established that LLM ensembles have substantial theoretical potential. For instance, the best model on the Defects4J benchmark, DeepSeek (16b), repairs 112 bugs while the ensemble of all models collectively could increase this number to 205, indicating a performance increase of 83%. However, this represents an idealized upper bound achievable only with the perfect selection oracle. Therefore, there is a need to study different selection mechanisms capable of selecting correct solutions from the pool of candidates generated by an ensemble.

## 4.2 Results of RQ2

Given a pool of outputs generated by the models of an ensemble, the challenge lies in selecting the most likely candidate outputs for validation, since testing all pool of outputs is often infeasible. We start this analysis by first establishing a baseline with a Naive strategy. The results from this approach are shown under the column *Naive* in Table 4, Table 5, and Table 6. The performance of this Naive heuristic varies across the benchmarks. We obtain the best results on LiveCodeBench, for instance, the ensemble of small models solved 142 problems, improving the best single model’s score (130), but still far from reaching the theoretical maximum of 185. This gap in performance decreases on HumanEval-Java, where the ensemble of large models solves 127 problems contrasting with the 122 problems solved by the best individual model. But, still far from the theoretical maximum of 139, while for other ensembles the results

barely improve the performance of the best model. For the more complex Defects4J benchmark, this strategy improves the score only for the ensemble of large models while decreasing it for the other two ensembles when compared to the performance of the best model. These results underscore the need for more sophisticated selection mechanisms.

**4.2.1 Outputs-based Selection Strategies.** We first focus on output-based metrics, which assess the similarity of an output to other candidates in the pool. The results are detailed in Table 4, Table 5, and Table 6. A key finding is the consistent failure of consensus-based selection. Across all benchmarks and ensembles, selecting the candidates with the highest CodeBLEU and CodeBERT scores results in poor performance, worse even than the Naive baseline approach. This phenomenon, which we term the *popularity trap*, suggests that models frequently produce syntactically similar but semantically incorrect solutions. In other words, models frequently fail in the same manner across their generated outputs. Relying on model consensus amplifies this phenomenon, often filtering out correct solutions for problems that not all models could solve.

Strategies based on disagreement (lowest scores) improve the performance substantially, as they avoid the popularity trap. However, this approach consistently outperforms the Naive heuristic only on the Defects4J benchmark. By relying on the lowest similarity, we are selecting any output with a hallucination that makes it significantly different from the rest of the generated outputs. Looking at the previous two approaches, we can agree that ensembles would benefit from a heuristic that does not fall into the popularity trap but neither allows arbitrary hallucinations. Therefore, we arrive at the third proposed strategy based on diversity.

The results clearly indicate that the diversity approach consistently improves the performance over the previous two approaches across both metrics and all three benchmarks. This heuristic proves itself most beneficial on the Defects4J benchmark, where the ensemble of all models selects a correct output for 164 out of 205 the-oretically solvable problems. This represents 80% of its theoretical potential, substantially exceeding the 97 correct outputs achieved by the Naive approach. On the other hand, on HumanEval-Java, the diversity strategy ranges from slightly improving the Naive approach by 3 on the ensemble of large models, to more substantial benefits on the ensemble of small models, where it realizes over 95% of the ensemble’s theoretical potential. Finally, in LiveCodeBench, the diversity strategy achieves approximately the same number of problems solved except for the ensembles of all models, where it decreases the number up to 9 compared to the Naive approach.

**4.2.2 Confidence-based Selection Strategies.** In contrast to output-based metrics, confidence-based metrics follow a more expected trend. For the three confidence-based metrics, lower values indicate that the output is more natural or that a model is more confident in predicting it. Thus, selecting outputs with lower scores indicates consensus between the models. We observe similar trends for NLL/Byte and Entropy/Byte, where consensus consistently achieves better scores than disagreement (selecting the highest) across benchmarks and ensembles. We suspect that using the model’s internal confidence scores helps avoid the popularity trap described in the previous subsection. However, Sum Entropy surprisingly does not follow the same trend for the two APR benchmarks. While the best

scores on the HumanEval-Java benchmark are lower compared to the other two metrics, we notice a substantial difference on the Defects4J benchmark. Specifically, in the latter benchmark, the best score from a consensus strategy is 123 problems solved by the ensemble of large models using Entropy/Byte, contrasting with the disagreement approach with Sum Entropy which solves 135 problems. Since this metric is length-dependent, selecting the highest scores can function similarly to selecting the lowest scores in similarity output-based metrics. This would explain the similar scores between the two approaches, especially the sudden increase in LiveCodeBench in the ensemble of large models. Therefore, although achieving higher scores than other confidence-based metrics, it suffers from similar limitations where an output containing a longer hallucination from the model would receive a higher score.

Similar to the output-based metrics, these confidence-based metrics benefit from a diversity heuristic across all benchmarks. On the Defects4J benchmark, the best score of 135 problems achieved with the previous strategies is increased to 157 by switching to this heuristic. Furthermore, on this benchmark, a new best score of 167 solved problems is achieved by increasing the size of the ensemble to all models. Diversity improves the best results across benchmarks, increasing from 123 to 135 solved problems in HumanEval-Java or improving from 130 to 170 problems solved in LiveCodeBench.

**4.2.3 Generalization to smaller ensembles.** While large ensembles show high potential, practical applications may need cost-effective solutions. Therefore, we study the generalization of the previous strategies to smaller ensembles. In particular, we analyze ensembles with two models by considering every possible pair of models. First, we investigate the effectiveness of our Naive baselines, which achieved high performance on ensembles of 5 and 10 models. Figure 3 shows heatmaps indicating the performance difference between the Naive strategy for any ensemble of two models and simply choosing the best single model in the pair. Negative values indicate cases where the Naive strategy performs worse than the best model of the pair. On Defects4J, these negative cases are frequent (17 out of 45), reaching a maximum drop of -15 problems when pairing  $CL_S$  and  $DS_L$ . Similarly, we also note drops of -8 and -6 problems for HumanEval-Java and LiveCodeBench, respectively. However, across all three benchmarks, we can also notice performance gains by using the Naive strategy. For instance, the combination of  $DS_L$  and  $QW_L$  results in an improvement of 22 problems for Defects4J. Similarly to the negative trend, this positive trend also extends to the other two benchmarks, resulting in an improvement of 12 and 17 problems solved in HumanEval-Java and LiveCodeBench, respectively. These results demonstrate that any ensemble is not always beneficial, and one needs to be careful when using a Naive ensemble strategy. Without studying the combinations of the models and without a complex selection strategy, combining models can be actively detrimental to the final outcomes.

**4.2.4 Complex strategies in smaller ensembles.** We apply one of our strategies to these same model pairs. We have chosen CodeBLEU but the other metrics provide very similar trends. As a selection strategy, we chose to maximize diversity since it provided the best outcomes on bigger ensembles regardless of the metric chosen (see Table 4, Table 5, and Table 6). Figure 4 shows heatmaps indicating the performance difference between our chosen approach and the

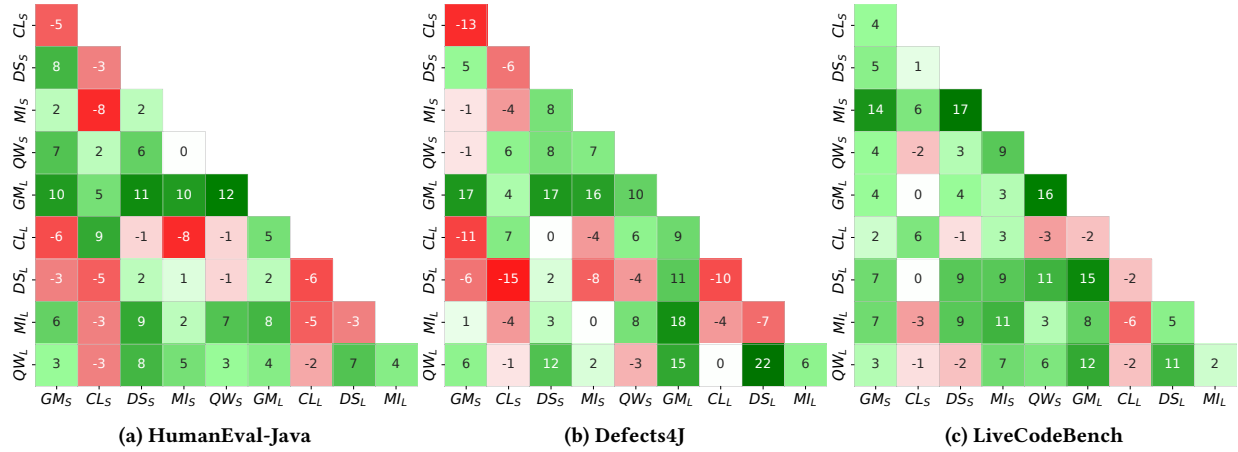


Figure 3: [RQ2.2] Comparison of two-model ensembles with Naive strategy vs. the best score of the two models. The heatmaps show the difference in the number of plausible candidates per benchmark (e.g. a negative score indicates that the Naive ensemble strategy performs worse than picking the better single model).

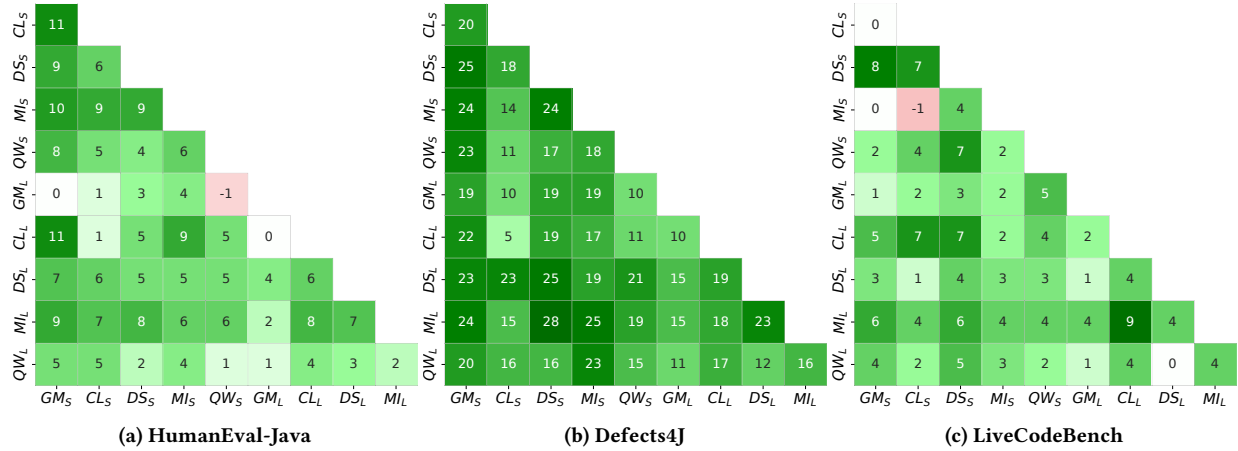


Figure 4: [RQ2.2] Comparison of two-model ensembles with diversity-based strategy using CodeBLEU vs. Naive strategy. The heatmaps show the difference in the number of plausible candidates per benchmark (e.g. a negative score indicates that the CodeBLEU diversity-based strategy performs worse than the Naive ensemble strategy).

Naive strategy. The results clearly indicate that our diversity-driven strategy provides a consistent and substantial performance gain across all model pairs and benchmarks. Our strategy outperforms or matches the Naive approach for all pairs except two, one case in HumanEval-Java and one case in LiveCodeBench, both cases decreasing the number of problems solved by only one. Furthermore, for all pairs of models and benchmarks, our strategy provides an improvement over the best score of the two models, consistently outperforming the Naive strategy. The improvements are most pronounced on Defects4J, with over 20 additional problems solved for 16 pairs compared to the Naive strategy. These results indicate that diversity-based selection is not only effective for larger ensembles, but it is even more effective for smaller ones. While we present results for only one strategy for brevity, these trends are consistent across other metrics and are available in our public repository.<sup>11</sup>

**4.2.5 Computational Overhead of Strategies.** The size of the ensemble is one of the key factors indicating the computational cost, however, it is not the only one. The metrics we have evaluated also add overhead to the calculations. Output-based metrics require pairwise comparison between all candidate solutions, which scales quadratically with the number of candidates. As the ensemble size grows, this cost may become a bottleneck. However, these metrics hold the advantage of being model-agnostic, as they use the generated code directly. Moreover, their hardware requirements are relatively low since CodeBLEU is CPU-bound and CodeBERTScore can be run effectively on consumer-grade GPUs.

In comparison, confidence-based metrics are more scalable. The cost of computing entropy scales linearly with the number of candidates. This linear efficiency makes them suitable for large ensembles

where quadratic scaling may be unfeasible. However, these metrics depend on the availability of token-level probabilities from the underlying model. This information is often not disclosed in closed-source models and some commercial APIs, rendering these metrics unfeasible.

## 5 Threats to Validity

One internal validity threat relates to the hyperparameters selection (e.g. in the generation of outputs or for the selection metrics). We address this by enforcing standard and deterministic generation settings with a temperature of 0 across all models and default hyperparameters for the selection metrics. Another threat to internal validity is concerned with potential data leakage from benchmarks into the training data of the models. This is particularly problematic for older benchmarks such as Defects4J. We address this threat by including a more recent APR benchmark (i.e. HumanEval-Java) and a code generation benchmark created specifically to avoid data contamination (i.e. LiveCodeBench).

A threat to the external validity of our study is the generalizability of results, for instance, to other models or ensembles, as well as other tasks and programming languages. We mitigate this threat by employing a diverse set of ten models from five different families and various parameter sizes. While exact performance numbers may differ, we believe that our core findings are likely to hold more broadly. However, confirming this across even bigger and more heterogeneous ensembles remains an important direction for future work. Similarly, our study uses Java and Python and focuses on code generation and repair, and further work would be needed to confirm our results for other tasks and languages.

The main threat to construct validity is our reliance on plausibility to assess correctness. Although practical and scalable, this metric does not guarantee true correctness since tests may not fully cover edge cases or outputs may overfit to the tests. While some prior work in the APR field manually checked all outputs, this is a labor-intensive and subjective process, as reviewers may decide correctness based on different standards [36]. In our experiments, we generate nearly 20,000 plausible candidate solutions which makes manual evaluation unfeasible. Another threat to construct validity relates to the number of outputs we choose to generate or select from the pool of candidates. To mitigate this threat, we select this based on observations from related work [15, 29, 33].

## 6 Conclusion

This work provides empirical evidence that an ensemble of diverse models, if combined effectively, outperforms individual models on two software engineering tasks. We systematically explore the complementarity of different LLMs and identify effective heuristics for selecting correct solutions from the aggregated pool of outputs.

Our extensive empirical study employed ten LLMs from five families across three benchmarks for code generation and automatic program repair. We provide evidence that smaller models can provide valuable solutions that were not found by their larger counterparts. Moreover, the likelihood of providing diverse solutions is increased if the models belong to different families. We find that complementarity can improve the ensemble’s theoretical potential over its best individual component by as much as 83% (from 112 to

205 solved problems on Defects4J). However, we demonstrate that the realization of the potential of the ensemble critically depends on the selection strategy. A naive selection strategy is unreliable, and heuristics based on consensus often converge on popular but incorrect solutions, a situation we define as the *popularity trap*. We introduce a diversity-based selection strategy that consistently outperforms consensus, achieving up to 95% of the ensemble’s theoretical potential. Finally, we extend our analysis to smaller and cost-effective ensembles of two models, where we empirically show that a diversity-based strategy consistently outperforms the results achieved by either of the models and the naive strategy.

Our findings lay the foundation for building more sophisticated and efficient ensembles. Each component can be selected based on its unique and non-overlapping capabilities. Furthermore, identifying the unique problem-solving capabilities of each model is key to advancing into more complex multi-agent systems. For practitioners, our work provides actionable insights: deploying an ensemble of models, especially smaller ones, with a diversity-based selection heuristic results in a consistent performance increase compared to relying on a single model.

**Future Work:** There are several ways to extend this work. First is expanding the scope and scale of our study as indicated in the previous section. This includes testing larger and more heterogeneous ensembles, investigating other software engineering tasks (e.g., code summarization, defect detection, or test case generation), and other programming languages. Another is to investigate alternative metrics and selection strategies. For example, a multi-stage process could use a cheap metric to quickly filter a large pool of candidates and then apply a more expensive metric on the reduced set.

Finally, instead of simply selecting from a static pool of generated outputs, models could be made to interact dynamically in an agentic fashion. This could, for example, involve (a) iterative refinement [8], where one or more models generate solutions, and other models debug these, creating an iterative refinement cycle, or (b) task decomposition, where a primary agent breaks down a complex coding problem into sub-tasks that are delegated to specialized models that are best suited for each part.

## 7 Data Availability

The temporary and anonymized replication package for this work is available online.<sup>11</sup> This package encompasses: (1) the source code required to replicate the experiments, (2) the generated outputs for all benchmarks and models, (3) the metrics calculated for said outputs, and (4) the scripts to analyze the results and generate figures.

## Acknowledgments

This work is supported by the Research Council of Norway through the secureIT project (IKTPLUSS #288787), and by the European Union through the Horizon Europe Marie Skłodowska-Curie Actions (#101151798). The empirical evaluation made use of the Experimental Infrastructure for Exploration of Exascale Computing (eX3), financially supported by the Research Council of Norway under contract #270053. In addition, we acknowledge Sigma2, Norway for awarding this project access to the LUMI supercomputer, owned

<sup>11</sup><https://figshare.com/s/bcb1bfd02cf5ef993e4e>

by the EuroHPC Joint Undertaking, hosted by CSC (Finland) and the LUMI consortium through the Research Council of Norway.

## References

- [1] Mari Ashiga, Wei Jie, Fan Wu, Vardan Voskanyan, Fateme Dinmohammadi, Paul Brookes, Jingzhi Gong, and Zheng Wang. 2025. Ensemble Learning for Large Language Models in Text and Code Generation: A Survey. *arXiv:2503.13505* [cs] doi:10.48550/arXiv.2503.13505
- [2] Jizheng Chen, Kounianhua Du, Xinyi Dai, Weiming Zhang, Xihuai Wang, Yasheng Wang, Ruiming Tang, Weinan Zhang, and Yong Yu. 2025. Debate-Coder: Towards Collective Intelligence of LLMs via Test Case Driven LLM Debate for Code Generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 12055–12065. doi:10.18653/v1/2025.acl-long.589
- [3] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, Yan Liu, Enyu Zhou, Ming Zhang, Yuhao Zhou, Yueming Wu, Rui Zheng, Ming Wen, Rongxiang Weng, Jingang Wang, Xunliang Cai, Tao Gui, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. *arXiv:2407.06153* [cs] doi:10.48550/arXiv.2407.06153
- [4] Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. 2021. Is a Single Model Enough? MuCoS: A Multi-Model Ensemble Learning Approach for Semantic Code Search. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. ACM, Virtual Event Queensland Australia, 2994–2998. doi:10.1145/3459637.3482127
- [5] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597503.3639219
- [6] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (Icse-Fose)*. 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008
- [7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139
- [8] Anastasiia Grishina, Vadim Liventsev, Aki Härmä, and Leon Moonen. 2025. Fully Autonomous Programming Using Iterative Multi-Agent Debugging with Large Language Models. *ACM Trans. Evol. Learn. Optim.* 5, 1 (March 2025), 8:1–8:37. doi:10.1145/3719351
- [9] Michael Hassid, Tal Remez, Jonas Gehring, Roy Schwartz, and Yossi Adi. 2024. The Larger the Better? Improved LLM Code-Generation via Budget Reallocation. *arXiv:2404.00725* [cs] doi:10.48550/arXiv.2404.00725
- [10] Junda He, Christoph Treude, and David Lo. 2025. LLM-based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* 34, 5 (May 2025), 124:1–124:30. doi:10.1145/3712003
- [11] Max Hort, Linas Vidziunas, and Leon Moonen. 2025. Semantic-Preserving Transformations as Mutation Operators: A Study on Their Effectiveness in Defect Detection. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE Computer Society, 337–346. doi:10.1109/ICSTW64639.2025.10962512
- [12] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8 (Dec. 2024), 220:1–220:79. doi:10.1145/3695988
- [13] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=chfJJYC3iL>
- [14] Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. LLM-blender: Ensembling Large Language Models with Pairwise Ranking and Generative Fusion. *arXiv:2306.02561* [cs] doi:10.48550/arXiv.2306.02561
- [15] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 1430–1442. doi:10.1109/ICSE48619.2023.00125
- [16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *23th International Symposium on Software Testing and Analysis (ISSTA) (ISSTA 2014)*. ACM, 437–440. doi:10.1145/2610384.2628055
- [17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv:2001.08361* [cs] doi:10.48550/arXiv.2001.08361
- [18] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 615–627. doi:10.1145/3377811.3380338
- [19] Yixin Liu and Pengfei Liu. 2021. SimCLS: A Simple Framework for Contrastive Learning of Abstractive Summarization. *arXiv:2106.01890* [cs] doi:10.48550/arXiv.2106.01890
- [20] Jinliang Lu, Ziliang Pang, Min Xiao, Yaochen Zhu, Rui Xia, and Jiajun Zhang. 2024. Merge, Ensemble, and Cooperate! A Survey on Collaborative Strategies in the Era of Large Language Models. *arXiv:2407.06089* doi:10.48550/ARXIV.2407.06089
- [21] Saeed Masoudnia and Reza Ebrahimpour. 2014. Mixture of Experts: A Literature Survey. *Artificial Intelligence Review* 42, 2 (Aug. 2014), 275–293. doi:10.1007/s10462-012-9338-y
- [22] Ibomoiye Domor Mienye and Yanxia Sun. 2022. A Survey of Ensemble Learning: Concepts, Algorithms, Applications, and Prospects. *IEEE Access* 10 (2022), 99129–99149. doi:10.1109/ACCESS.2022.3207287
- [23] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2228–2240. doi:10.1145/3510003.3510040
- [24] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Pierre Isabelle, Eugene Charniak, and Dekang Lin (Eds.). Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. doi:10.3115/1073083.1073135
- [25] Zeeshan Rasheed, Muhammad Waseem, Kai Kristian Kemell, Aakash Ahmad, Malik Abdul Sami, Jussi Rasku, Kari Systä, and Pekka Abrahamsson. 2025. Large Language Models for Code Generation: The Practitioners Perspective. *arXiv:2501.16998* [cs] doi:10.48550/arXiv.2501.16998
- [26] Mathieu Ravaut, Shafiq Joty, and Nancy F. Chen. 2023. SummaReranker: A Multi-Task Mixture-of-Experts Re-ranking Framework for Abstractive Summarization. *arXiv:2203.06569* [cs] doi:10.48550/arXiv.2203.06569
- [27] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: A Method for Automatic Evaluation of Code Synthesis. *arXiv:2009.10297* [cs] doi:10.48550/arXiv.2009.10297
- [28] Shahriyar Zaman Ridoy, Md. Shazzad Hossain Shaon, Alfredo Cuzzocrea, and Mst Shapna Akter. 2024. EnStack: An Ensemble Stacking Framework of Large Language Models for Enhanced Vulnerability Detection in Source Code. In *2024 IEEE International Conference on Big Data (BigData)*. 6356–6364. doi:10.1109/BigData64233.2024.10825609
- [29] Fernando Vallecillos Ruiz, Max Hort, and Leon Moonen. 2025. The Art of Repair: Optimizing Iterative Program Repair with Instruction-Tuned Models. *arXiv:2505.02931* [cs] doi:10.48550/arXiv.2505.02931
- [30] Julian Salazar, Davis Liang, Toan Q. Nguyen, and Katrin Kirchhoff. 2020. Masked Language Model Scoring. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2699–2712. *arXiv:1910.14659* [cs] doi:10.18653/v1/2020.acl-main.240
- [31] Thibault Sellam, Dipanjan Das, and Ankur P. Parikh. 2020. BLEURT: Learning Robust Metrics for Text Generation. *arXiv:2004.04696* [cs] doi:10.48550/arXiv.2004.04696
- [32] Alexander Shypula, Shuo Li, Botong Zhang, Vishakh Padmakumar, Kayo Yin, and Osbert Bastani. 2025. Evaluating the Diversity and Quality of LLM Generated Content. In *Second Conference on Language Modeling*. <https://openreview.net/forum?id=O7bF6nLSOD#discussion>
- [33] André Silva, Sen Fang, and Martin Monperrus. 2025. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. *IEEE Transactions on Software Engineering* 51, 8 (Aug. 2025), 2366–2380. doi:10.1109/TSE.2025.3581062
- [34] Elliott Sober. 2015. *Ockham's Razors: A User's Manual*. Cambridge University Press, Cambridge. doi:10.1017/CBO9781107705937
- [35] Zhihong Sun, Jia Li, Yao Wan, Chuanyi Li, Hongyu Zhang, Zhi Jin, Ge Li, Hong Liu, Chen Lyu, and Songlin Hu. 2025. Ensembling Large Language Models for Code Vulnerability Detection: An Empirical Evaluation. *arXiv:2509.12629* [cs] doi:10.48550/arXiv.2509.12629

- [36] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2021. Automated Patch Correctness Assessment: How Far Are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 968–980. doi:10.1145/3324884.3416590
- [37] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddartha Venkat Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2024. LiveBench: A Challenging, Contamination-Limited LLM Benchmark. In *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=sKYHBTaxVa>
- [38] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 1482–1494. doi:10.1109/ICSE48619.2023.00129
- [39] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Singapore Singapore, 959–971. doi:10.1145/3540250.3549101
- [40] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Zizheng Zhang, Haotian Zhang, and Yuqun Zhang. 2024. How Far Can We Go with Practical Function-Level Program Repair? arXiv:2404.12833 [cs] doi:10.48550/arXiv.2404.12833
- [41] Tengfei Xue, Xuefeng Li, Tahir Azim, Roman Smirnov, Jianhui Yu, Arash Sadrieh, and Babak Pahlavan. 2024. Multi-Programming Language Ensemble for Code Generation in Large Language Model. arXiv:2409.04114 [cs] doi:10.48550/arXiv.2409.04114
- [42] Aidan Z. H. Yang, Sophia Kolak, Vincent J. Hellendoorn, Ruben Martins, and Claire Le Goues. 2024. Revisiting Unnaturalness for Automated Program Repair in the Era of Large Language Models. arXiv:2404.15236 [cs] doi:10.48550/arXiv.2404.15236
- [43] Weizhe Yuan, Graham Neubig, and Pengfei Liu. 2021. BARTScore: Evaluating Generated Text as Text Generation. In *Advances in Neural Information Processing Systems*, Vol. 34. Curran Associates, Inc., 27263–27277. [https://proceedings.neurips.cc/paper\\_files/paper/2021/hash/e4d2b6e6fdeca3e60e0f1a62fee3d9dd-Abstract.html](https://proceedings.neurips.cc/paper_files/paper/2021/hash/e4d2b6e6fdeca3e60e0f1a62fee3d9dd-Abstract.html)
- [44] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating Text Generation with BERT. arXiv:1904.09675 [cs] doi:10.48550/arXiv.1904.09675
- [45] Wenkang Zhong, Chuanyi Li, Kui Liu, Tongtong Xu, Jidong Ge, Tegawende F. Bissyande, Bin Luo, and Vincent Ng. 2024. Practical Program Repair via Preference-Based Ensemble Strategy. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–13. doi:10.1145/3597503.3623310
- [46] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 13921–13937. doi:10.18653/v1/2023.emnlp-main.859