Scaling Laws for Code: Every Programming Language Matters

Anonymous ACL submission

Abstract

Large language models (LLMs) are powerful but costly to train, with scaling laws predicting performance from model size, data, and compute. However, different programming languages (PLs) have varying impacts during pre-training that significantly affect base model performance, leading to inaccurate performance prediction. Existing works focus on language-agnostic settings, neglecting the inherently multilingual nature of modern software development. Therefore, it is first necessary to investigate the scaling laws of different PLs, and then consider their mutual influences to arrive at the final multilingual scaling law. In this paper, we present the first systematic exploration of scaling laws for multilingual code pre-training, conducting over 1000+ experiments (Equivalent to 336,000+ H800 hours) across multiple PLs, model sizes (0.2B to 14B parameters), and dataset sizes (1T tokens). We establish scaling laws for code LLMs, showing interpreted languages benefit more from scale than compiled ones. Multilingual pre-training provides synergistic benefits between similar languages, with parallel pairing enhancing cross-lingual abilities. We propose a proportion-dependent scaling law that optimally allocates training tokens by prioritizing high-utility languages (e.g., Python), balancing high-synergy pairs (e.g., JavaScript-TypeScript), and reducing fast-saturating languages (Rust), achieving superior performance versus uniform distribution.

1 Introduction

Code large language models (code LLMs) have achieved excellent coding performance in multiple programming languages (PLs), guided by the scaling law in general domains (Chen et al., 2021; Hui et al., 2024; Guo et al., 2024; Wang et al., 2024; Bi et al., 2024; Hurst et al., 2024). Code LLMs significantly enhance developer productivity (Anysphere Inc., 2025), but training top-tier LLMs consumes enormous computing resources and costs (Kaplan et al., 2020; Hoffmann et al., 2022; Brown et al., 2020; Bi et al., 2024), making it prohibitively expensive to conduct ablation experiments on data composition or pre-training strategies at scale. This cost barrier limits our ability to systematically understand the key factors driving code LLM performance and hinders the development of more efficient training methodologies.

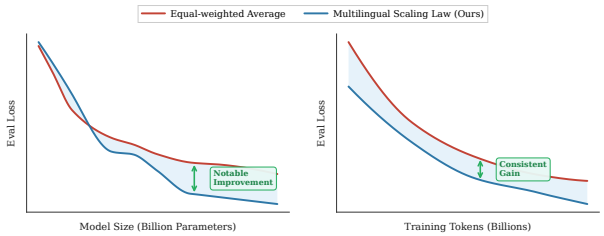

Figure 1: Evaluation loss comparison showing that the proposed multilingual scaling law achieves lower loss than the baseline across model sizes and token budgets.

Scaling laws characterize how model performance depends on model size, dataset size, and compute budget (Kaplan et al., 2020; Hoffmann et al., 2022). Recent work (Luo et al., 2025) extended these laws to code, showing that code pre-training is more data-hungry than natural language pre-training. However, modern software development is inherently multilingual, with developers working across Python, Java, and other PLs for diverse downstream tasks (e.g., OSAgent (Hu et al., 2025)). This raises critical questions: *(1) What is the scaling law for multilingual code pre-training and cross-lingual capabilities? (2) What is the optimal strategy for allocating training resources across different programming languages?*

To address this gap, we conduct the first systematic exploration of scaling laws for multilingual code pretraining. Our study comprises over 1000+ experiments spanning multiple PLs, model sizes (0.2B to 14B parameters), and dataset sizes (1T tokens), where all base models are pre-trained

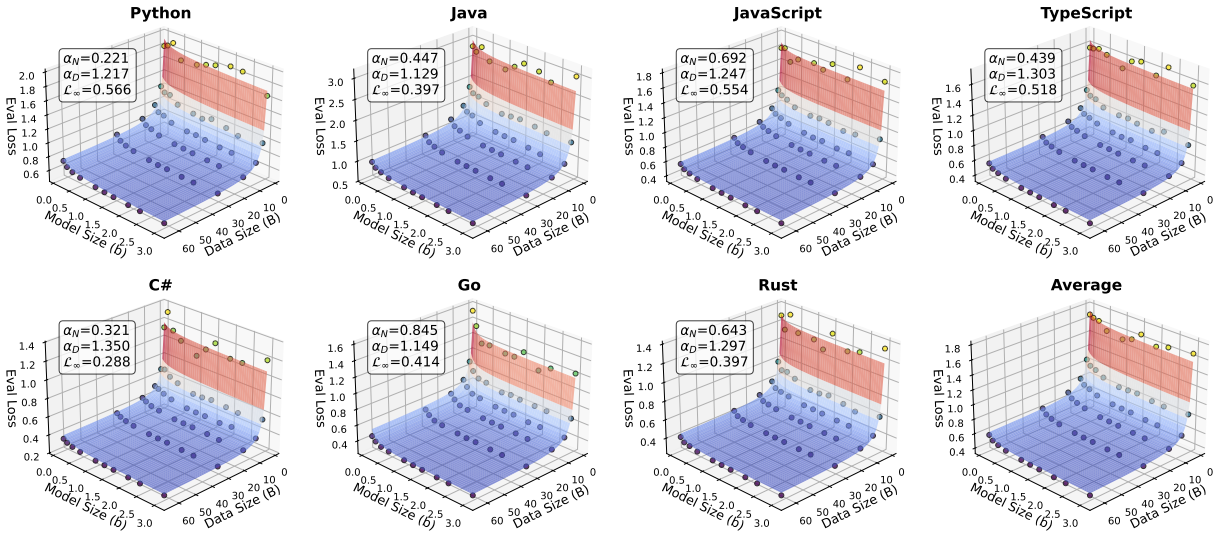

Figure 2: Scaling Laws for each PL independently. It shows a clear ordering of intrinsic predictability across PLs: C# < Java ≈ Rust < Go < TypeScript < JavaScript < Python.

from scratch. In Figure 1, our multilingual scaling law consistently outperforms the equal-weighted baseline across different model scales and training regimes. We investigate four fundamental aspects: (1) the trade-off between language diversity and language depth, (2) cross-lingual transfer effects in multilingual pre-training, (3) scaling dynamics for code translation tasks, and (4) optimal language proportion allocation strategies. Our findings reveal insights about the interplay between linguistic diversity and model performance, providing actionable guidance for training multilingual code LLMs.

Our key contributions include: (1) We establish language-specific scaling laws for each PL, revealing that interpreted PLs benefit more from scale than compiled PLs. The irreducible loss metric shows a complexity ordering (C# < Java ≈ Rust < Go < TypeScript < JavaScript < Python), where strictly-typed PLs are more learnable than dynamic ones. (2) We study synergy gains between PLs, showing that syntactically similar languages (e.g., Java-C#) enable positive transfer. Most PLs benefit from multilingual pre-training, indicating that optimal PL mixing must be tailored to each language's characteristics. (3) We investigate how data organization affects cross-lingual abilities. Experiments show that parallel pairing (concatenating translated code snippets) significantly outperforms baselines on multilingual translation and generation, with favorable scaling properties for learning cross-lingual alignments. (4) We propose a proportion-dependent multilingual scaling law with language-specific parameters and cross-lingual transfer effects. Under optimal allocation, the model prioritizes high-utility PLs (e.g., Python) and high-synergy pairs (e.g., JS-TS) while reducing tokens for fast-saturating PLs (e.g., Rust), achieving higher average performance across all PLs without degrading any single language.

2 Scaling Laws for Code Pre-training

2.1 ChinChilla Scaling Law

Scaling laws provide a theoretical framework for understanding how model performance evolves with computational resources. For LLM, the relationship between validation loss $\mathcal{L}$ and key factors (model parameters N, training tokens D, and compute budget C) can be characterized by power-law formulations (Hoffmann et al., 2022) as below:

$$\mathcal{L}(N, D) = \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{D_c}{D}\right)^{\alpha_D} + \mathcal{L}_\infty \quad (1)$$

where N is the number of model parameters, D is the dataset size (number of tokens), and C is the compute budget (FLOPs). N_c and D_c are scaling constants. α_N and α_D are power law exponents, and L_∞ is the irreducible loss (the inherent error that no model can eliminate).

2.2 Motivation and Formulation

Our goal is to address three fundamental questions about multilingual code pre-training: **(1) Language-Specific Scaling Dynamics:** How does each programming language scale with model size and data? Do different languages exhibit distinct scaling exponents and irreducible losses? **(2) Cross-Lingual Synergy Effects:** What synergis-

tic or antagonistic effects arise when mixing languages during pre-training? Can bilingual training outperform monolingual baselines under fixed compute budgets? **(3) Compositional Cross-Lingual Transfer:** Can models trained on supervised language pairs generalize to unseen pairs? Do different data organization strategies enable zero-shot cross-lingual transfer?

To address these questions, we conduct three complementary experimental studies spanning over 1000+ training runs across diverse model scales (0.2B to 14B parameters) and data volumes (up to 1T tokens). Our approach isolates specific variables while controlling for confounding factors, enabling the derivation of actionable scaling laws for practical multilingual code pre-training.

3 Language-specific Scaling Law

Language-specifc Scaling Law

(1) For code across all PLs, scaling up data size yields greater performance gains than scaling up model size.
(2) Different PLs exhibit distinct convergence rates during training and vary significantly in their inherent training difficulty.

3.1 Experimental Setting

We first try to establish baseline scaling laws for each programming language in isolation, investigating whether different programming languages exhibit distinct scaling exponents (α_N, α_D). and how the irreducible loss L_∞ varies across languages and what this reveals about intrinsic language complexity. We select 7 PLs that span diverse paradigms and application domains, including Python, Java, JavaScript, TypeScript, C#, Go, and Rust. For each PL, we train LLMs with different model parameters (different model sizes of 10 settings: 0.1B, 0.2B, 0.4B, 0.6B, 1.1B, 1.3B, 1.6B, 2.0B, 2.4B, 3.1B) and token budgets (budget training tokens of 6 settings: 2B, 4B, 8B, 16B, 32B, and 64B tokens), yielding 10 (model sizes) $\times$ 6 (token budgets) $\times$ 7 (PLs) = 420 experiments in total. These experiments provide dense coverage of the (N, D) space, enabling robust estimation of Chinchilla scaling law formulations for each PL.

To ensure a fair comparison across languages, all LLMs share the same model architecture, similar to LLaMA-2, including SwiGLU activations, rotary position embeddings (RoPE), multi-head attention (MHA), and RMSNorm . We collect a high-quality training corpus spanning multiple languages, where Python and other PLs are parallel corpus.

3.2 Language-Specific Scaling Laws

To establish the scaling laws for each PL, we train 420 LLMs across 7 PLs with different model sizes (N) and training tokens (D). For each PL, we fit both the Chinchilla-style power law.

Scaling Exponents and Optimal Allocation Figure 2 summarizes the fitted scaling parameters for each PL, revealing significant heterogeneity in scaling behaviors across languages. Interpreted languages exhibit larger scaling exponents than compiled languages. Python, as a dynamically-typed interpreted language, obtains the highest α_N and α_D values, benefiting more from increases in both model parameters and training data. In contrast, Rust shows notably smaller exponents due to its explicit type annotations and rigid syntactic structure, making it more learnable with fewer parameters and less data. The optimal N:D ratios also vary substantially: PLs with higher α_D relative to α_N (e.g., Python) favor larger datasets, while those with lower α_D (e.g., Rust) achieve comparable performance with fewer tokens but may benefit from increased model capacity.

Irreducible Loss and Language Complexity The irreducible loss $\mathcal{L}_\infty$ measures language complexity as the lower bound on achievable perplexity with infinite compute. Our results show: C# $<$ Java $\approx$ Rust $<$ Go $<$ TypeScript $<$ JavaScript $<$ Python. C# achieves the lowest $\mathcal{L}_\infty$ due to its strict type system and standardized ecosystem. Java and Rust enforce strong constraints limiting expression diversity. Go's minimalist design yields moderate predictability, while TypeScript retains JavaScript's unpredictability through optional typing. JavaScript's high $\mathcal{L}_\infty$ stems from dynamic typing and flexible paradigms. Python exhibits the highest $\mathcal{L}_\infty$, reflecting its dynamic nature and diverse coding styles.

4 Language Mixture for Data Scarcity

Language-specifc Scaling Law

(1) For code data across all programming languages, scaling up data size yields greater performance gains than scaling up model size.
(2) Different PLs exhibit distinct convergence rates during training. PLs vary significantly in their inherent training difficulty.

4.1 Experimental Setting

The second experiment investigates the effects of mixing two PLs during pre-training, examining whether mixing with a different PL improves performance compared to pre-training on a single PL. We use the total training budget of 128B tokens and compare different data compositions for each PL. For a given target PL L_i (e.g., Python), we construct a baseline training setting by repeating D_{L_i} ($|D_{L_i}| = 64$ B) twice and obtain a total of 128B tokens. Then, we mix the PL L_i ($|D_{L_i}| = 64$B) and PL L_j ($|D_{L_j}| = 64$B) to study the language interference between L_i and L_j. In summary, we compare pre-training setting (1): $D_{L_i} + D_{L_i}$ and setting (2): $D_{L_i} + D_{L_j}$. After pre-training, LLM is only evaluated on downstream tasks in the target language L_i to study the interference of L_j to L_i. For example, an LLM trained on "Python + Java" is evaluated on Python validation loss. This setup allows us to quantify the benefit (or harm) of including auxiliary language data. We define the synergy gain as:

$$\Delta(L_i, L_j) = \mathcal{L}(L_i + L_j) - \mathcal{L}(L_i + L_i) \quad (2)$$

where $\mathcal{L}(L_i + L_j)$ denotes the validation loss trained on $D_{L_i} + D_{L_j}$ while $\mathcal{L}(L_i + L_i)$ denotes the validation loss trained on $D_{L_i} + D_{L_i}$. A positive Δ indicates that mixing with PL L_j improves performance on L_i compared to the self-repetition baseline. For most low-resource PLs, providing auxiliary PLs can significantly enhance low-resource language performance.

4.2 Bilingual Mixture Effects

We train 28 LLMs with different mixing PLs ($\frac{7\times6}{2} + 7 = 28$) with a fixed architecture ranging from 0.1B to 3.1B parameters and 128B total tokens.

Synergy Gain Matrix Table 1 presents the synergy gain $\Delta(\mathcal{L}_i, \mathcal{L}_j)$ for all language pairs, where positive values indicate performance improvements over the self-repetition baseline ($D_{L_i} \times 2$). Our results reveal two key findings: (1) Multilingual pre-training provides substantial benefits, with 6/7 languages showing consistent positive synergy across auxiliary language combinations. (2) PLs sharing similar syntax, semantics, or paradigms benefit significantly from joint training through transferred representations. Java exhibits exceptional synergy gains with all auxiliary PLs, particularly C# ($\Delta = 0.186$), JavaScript ($\Delta = 0.114$), TypeScript ($\Delta = 0.109$), and Rust ($\Delta = 0.112$). The Java-C# pair likely benefits from their shared object-oriented paradigm and design patterns. However, Python shows small negative effects with most auxiliary PLs: JavaScript ($\Delta = -0.009$), TypeScript ($\Delta = -0.007$), C# ($\Delta = -0.013$), Go ($\Delta = -0.016$), and Rust ($\Delta = -0.021$), with only Java providing modest positive synergy ($\Delta = 0.010$). Notably, this transfer is asymmetric: Python as an auxiliary language benefits other targets (e.g., $\Delta = 0.054$ for Java), but Python itself suffers when mixed with them. Overall, multilingual pre-training benefits most PL, though Python may require tailored mixture strategies.

Suggestions for Data Curation These findings have direct implications for constructing multilingual code corpora. When training tokens are limited, mixing syntactically-related PLs outperforms naively upsampling a single PL. The positive synergy effects suggest that linguistic diversity acts as data augmentation that improves model robustness. For realistic multilingual pre-training, a mixed-language training regime is superior to language-specific fine-tuning. However, the optimal mixing ratio remains an open question. While this section uses a 50:50 ratio, section 6 will investigate whether asymmetric mixtures (e.g., 75:25) yield further improvements.

5 Cross-Lingual Scaling Laws

Cross-Lingual Pre-training Strategies

Data: Parallel data for `Python`↔Others pairs, plus monolingual data for all PLs. (1) **Baseline:** Train on shuffled monolingual data; Test on all directions.
(2) **Supervised:** Train on Python↔Others; Test on Python↔Others.
(3) **Zero-Shot:** Train on Python↔Others; Test on Non-Python pairs (e.g., Java→C#).

Language	Python	Java	JavaScript	TypeScript	C#	Go	Rust
Python	0.75	0.76 (↑**1.36%**)	0.77 (↓**1.12%**)	0.74 (↓**0.95%**)	0.77 (↓**1.69%**)	0.76 (↓**2.13%**)	0.77 (↓**2.72%**)
Java	0.85 (↑**6.02%**)	0.79	0.79 (↑**12.62%**)	0.90 (↑**12.08%**)	0.81 (↑**20.58%**)	0.79 (↑**10.68%**)	0.72 (↑**12.41%**)
JavaScript	0.51 (↑**5.49%**)	0.52 (↑**2.98%**)	0.53	0.53 (↑**4.69%**)	0.54 (↑**2.44%**)	0.54 (↑**1.34%**)	0.53 (↑**2.56%**)
TypeScript	0.51 (↑**4.17%**)	0.53 (↑**2.29%**)	0.53 (↑**3.34%**)	0.52	0.53 (↑**1.68%**)	0.52 (↑**1.39%**)	0.53 (↑**1.18%**)
C#	0.33 (↑**3.84%**)	0.33 (↑**1.93%**)	0.34 (↑**3.10%**)	0.34 (↑**3.71%**)	0.34	0.34 (↑**1.87%**)	0.35 (↑**1.98%**)
Go	0.41 (↑**4.77%**)	0.41 (↑**2.95%**)	0.42 (↑**2.70%**)	0.42 (↑**4.41%**)	0.43 (↑**2.95%**)	0.42	0.42 (↑**2.86%**)
Rust	0.38 (↑**3.87%**)	0.38 (↑**2.89%**)	0.40 (↑**4.20%**)	0.38 (↑**4.05%**)	0.38 (↑**2.81%**)	0.38 (↑**2.80%**)	0.38

Table 1: Synergy gain matrix (Reordered). Values indicate absolute performance, while parentheses show relative improvement vs baseline. **Bold numbers** indicate the percentage change. Background intensity indicates magnitude (Darker Red = Higher Gain).

5.1 Experimental Setting

Training Corpus We construct a comprehensive multilingual corpus of 900B tokens containing algorithmically equivalent implementations across seven programming languages, where Python serves as a pivot language with parallel implementations to six target languages (Java, JavaScript, TypeScript, C#, Go, and Rust). Notably, direct pairs between non-Python languages are absent from the training data. We augment this with 100B tokens from FineWeb-Edu for natural language understanding, yielding 1T total tokens. We train LLMs with different model parameters (different model sizes of 10 settings: 0.1B, 0.2B, 0.4B, 0.6B, 1.1B, 1.3B, 1.6B, 2.0B, 2.4B, 3.1B) and token budgets (budget training tokens of 6 set- tings: 2B, 4B, 8B, 16B, 32B, and 64B tokens) We systematically evaluate two data organization paradigms across five model scales (0.2B, 0.5B, 1.5B, 3B, and 7B parameters), training each configuration for a full epoch over the 1T token corpus, comprised of 900B code tokens and 100B natural language tokens for natural language understanding.

Cross-lingual Evaluation We construct an evaluation set to evaluate PL translation task. Three software engineers select 50 Python files from GitHub, ensuring that each code snippet is functionally translatable to all target PLs and the samples span diverse algorithmic tasks to avoid evaluation bias toward specific programming paradigms. Human annotators then manually produce equivalent implementations for 6 target PLs (Java, JavaScript, TypeScript, C#, Go, and Rust), following strict guidelines to preserve semantic equivalence while adhering to language-specific idioms. The resulting evaluation set comprises $50 \times A_7^2 = 2{,}100$ translation instances covering all 42 translation directions, with an average sequence length of 464 tokens. This comprehensive coverage enables systematic evaluation of both seen translation directions (Python $\leftrightarrow$ Others) and unseen zero-shot directions (Non-Python $\leftrightarrow$ Non-Python). Given the source code x of PL L_i, we calculate the loss $-\mathbb{E}[\log P(y|x)]$ of the target code y of PL L_j.

5.2 Pre-training on Unsupervised Data

Zero-shot Scaling Law Since the standard pre-training provides no direct alignment between language pairs, we evaluate LLM on code translation loss that require implicit cross-lingual capabilities. The zero-shot scaling follows:

$$\mathcal{L}_z(N) = A_z \cdot N^{-\alpha_z} + B_z \cdot D^{-\beta_z} + \mathcal{L}_{\infty,z} \quad (3)$$

where $A_z = 0.1574$, $B_z = 9.553$, $\alpha_z = 0.3470$, $\beta_z = 0.8829$, and $L_{\infty,z} = 0.1236$. Even without explicit supervision, LLMs develop emergent cross-lingual capability, suggesting that unsupervised pre-training can get the basic cross-lingual capability.

5.3 Pre-training on Supervised Data

Unlike pre-training without the explicit cross-lingual alignment, we concatenate the code snippet x of PL L_i and the corresponding translation y of PL L_j as (x, y), which provides an explicit document-level alignment signal.

Translation Scaling Law For language pairs explicitly aligned during training (Python $\leftrightarrow$ {Java, JavaScript, TypeScript, C#, Go, Rust}), we observe enhanced cross-lingual performance that scales as:

$$\mathcal{L}_a(N) = A_a \cdot N^{-\alpha_a} + B_a \cdot D^{-\beta_a} + \mathcal{L}_{\infty,a} \quad (4)$$

where $A_a = 0.0508$, $B_a = 0.793$, $\alpha_a = 6.404$, $\beta_a = 0.8829$, and $L_{\infty,a} = 0.1006$. The high scaling exponent α_z indicates that parallel pairing enables efficient exploitation of model capacity for learning cross-lingual alignment between seen language pairs.

	Python	Java	Go	C#	Javascript	Typescript	Rust
Python	–	0.71, 0.87, 0.11, 0.25	0.56, 1.45, 0.20, 0.28	0.62, 1.36, 0.19, 0.29	0.68, 1.45, 0.14, 0.38	0.61, 1.70, 0.19, 0.36	0.38, 2.40, 0.36, 0.46
Java	0.80, 0.89, 0.07, 0.36	–	0.69, 1.22, 0.15, 0.30	0.53, 2.92, 0.22, 0.54	0.37, 1.35, 0.28, 0.46	0.59, 1.11, 0.16, 0.35	0.43, 1.45, 0.28, 0.21
Go	0.18, 0.72, 0.38, 0.31	0.61, 0.80, 0.08, 0.17	–	0.53, 0.89, 0.15, 0.16	0.70, 0.81, 0.11, 0.24	0.10, 8.31, 0.72, 1.05	0.04, 0.65, 1.24, 0.39
C#	0.19, 0.90, 0.38, 0.09	0.51, 1.85, 0.15, 0.46	0.81, 1.73, 0.12, 0.52	–	0.14, 5.73, 0.68, 0.89	0.16, 5.15, 0.74, 0.85	0.04, 4.70, 1.32, 0.87
Javascript	0.47, 0.77, 0.17, 0.11	0.63, 0.74, 0.10, 0.19	0.61, 1.35, 0.17, 0.28	0.55, 1.06, 0.19, 0.23	–	0.33, 11.88, 0.47, 0.98	0.30, 1.04, 0.31, 0.10
Typescript	0.58, 0.82, 0.12, 0.36	0.81, 1.33, 0.08, 0.51	0.63, 1.01, 0.14, 0.24	0.23, 0.95, 0.49, 0.10	0.56, 40.06, 0.21, 1.20	–	0.11, 1.20, 0.79, 0.08
Rust	0.56, 0.81, 0.13, 0.14	0.56, 1.07, 0.14, 0.20	0.68, 1.07, 0.14, 0.27	0.43, 1.02, 0.17, 0.14	0.57, 0.86, 0.16, 0.17	0.27, 2.40, 0.31, 0.66	–

Table 2: Chinchilla scaling law parameters $(A, B, \alpha_N, \alpha_D)$ for baseline model across translation directions. Formula: $\mathcal{L}(N, D) = A/N^{\alpha_N} + B/D^{\alpha_D} + L_\infty$.

Zero-shot Translation Scaling Law Document-level pairing also improves zero-shot performance on unseen language pairs (e.g., Java ↔ Go, Rust ↔ JavaScript). Despite never observing direct alignments between non-Python languages, models exhibit compositional generalization:

$$\mathcal{L}_{zt}(N) = A_{zt} \cdot N^{-\alpha_{zt}} + B_{zt} \cdot D^{-\beta_{zt}} + \mathcal{L}_{\infty,zt} \quad (5)$$

where $A_{zt} = 0.0350$, $B_{zt} = 4.518$, $\alpha_{zt} = 0.781$, $\beta_{zt} = 0.869$, and $L_{\infty,zt} = 0.0524$. Zero-shot performance under document-level pairing substantially exceeds the shuffled baseline, suggesting the model uses Python as an implicit bridge through learned bidirectional mappings (e.g., Java → Python → Go). Table 2 shows that parallel pairing yields superior scaling performance for both seen and unseen translation directions.

5.4 Cross-Lingual Translation Strategies

We examine how different data organization strategies during pre-training affect the ability of LLM to perform cross-lingual code translation. We compare two strategies, including (1) random shuffling and (2) parallel pairing, across five LLM sizes (0.2B to 7B parameters) on 1T tokens.

Performance on Code Translation Figure 4 reports the average validation loss on the 12 seen translation directions (Python ↔ {Java, JavaScript, TypeScript, C#, Go, Rust}). As expected, the parallel pairing strategy achieves better performance compared to the random shuffling strategy. The parallel pairing acts as a soft alignment signal to enforce the LLM to learn the cross-lingual alignment. To further evaluate the cross-lingual capability, we evaluate all LLMs on multilingual code generation. Figure 4 explores two strategies of code concatenation, including direct concatenation $(x + y)$ and prompt-based concatenation (Please translate the following Python code to PL:\n \n x + Answer:y).

Performance on Code Generation Table 3 presents the evaluation results on the multilingual code generation benchmark MultiPL-E. The LLM trained with parallel pairing also gets the better multilingual code generation performance This suggests that document-level pairing is the optimal data organization strategy for multilingual code pre-training, balancing translation competence with general-purpose code understanding.

PL	0.5B	1.5B	3B	7B
Python	14.02 / 12.20	19.51 / 21.34	21.95 / 25.61	34.15 / 26.22
Java	1.90 / 8.23	5.70 / 15.84	6.96 / 22.78	14.56 / 32.28
JavaScript	7.45 / 7.45	21.74 / 19.89	24.22 / 24.84	36.02 / 37.27
TypeScript	15.72 / 13.21	22.64 / 21.93	29.56 / 28.30	40.25 / 40.25
C#	10.13 / 9.49	15.19 / 13.35	25.32 / 24.05	37.34 / 32.91
Average	9.84 / 10.12	16.96 / 18.47	21.60 / 25.12	32.46 / 33.79

Table 3: Evaluation results on multilingual code generation benchmark MultiPl-E (baseline/parallel pairing).

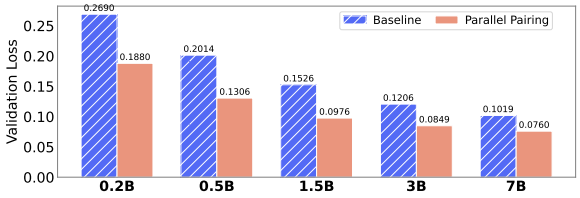

Figure 3: Validation loss on unseen translation directions. Each entry is the average loss across 30 translation pairs not seen during pre-training.

Zero-Shot Translation on Unseen Directions A key question is whether LLMs can generalize to translation directions not seen during pre-training, particularly between non-Python language pairs (e.g., Java → Go, Rust → JavaScript). Figure 4 reports validation loss on the 30 unseen translation directions. Both strategies demonstrate zero-shot translation capability on unseen directions. For the random shuffling strategy, it performs poorly on seen directions, but does not completely fail on unseen pairs, which can generate syntactically plausible translations, albeit with higher error rates. This suggests that the model learns some general notion of algorithmic equivalence across languages, even without explicit alignment during training. LLMs trained on parallel PLs achieve better performance on unseen directions compared to the baseline.

Scaling Laws for Code Translation To quantify how translation performance scales with model size

and data, we fit power-law curves to the validation loss on both seen and unseen translation directions. In Equation 4 and Equation 5, the fitted exponents α reveal how efficiently each strategy leverages additional model capacity. Strategies with higher α benefit more from scaling, while lower L_∞ indicates better asymptotic performance. These scaling laws enable practitioners to predict translation quality at scales beyond what was experimentally evaluated, informing decisions about model size and training compute allocation.

6 Guideline for Code Pre-training

Cross-Lingual Pre-training Strategies

(1) Uniform allocation is suboptimal (modest adjustments yield measurable gains).

(2) Language synergy effects are substantial and should guide corpus design.

6.1 Experimental Setup

We train two 1.5B parameter models with different training data distributions (400B tokens: 350B code + 50B FineWeb-Edu): (1) **Baseline (Uniform Allocation):** Equal allocation of 50B tokens to each PL (350B code + 50B FineWeb-Edu = 400B total), representing standard multilingual pre-training practice. (2) **Optimized (Guided Allocation):** Strategic allocation of the same 350B code tokens based on fitted scaling laws, synergy matrix, and language complexity analysis (350B code + 50B FineWeb-Edu = 400B total). Figure 5 presents the detailed token distribution for both strategies. The optimized allocation redistributes tokens based on marginal utility: more for high-α_D PLs (Python), balanced allocation for high-synergy pairs, and reduced tokens for fast-saturating PLs.

Proportion-dependent Multilingual Scaling Law Traditional scaling laws treat multilingual code as homogeneous, but PLs contribute differently to performance. We extend this by incorporating language proportions $p = (p_1, \ldots, p_K)$ explicitly:

$$\mathcal{L}(N, D; p) = A \cdot N^{-\alpha_N(p)} + B \cdot D_x^{-\alpha_D(p)} + L_\infty(p) \quad (6)$$

where $\alpha_N(p) = \sum_k p_k \alpha_N^k$, $\alpha_D(p) = \sum_k p_k \alpha_D^k$, and $L_\infty(p) = \sum_p p L_\infty^k$ are proportion-weighted averages of language-specific parameters from Figure 2. The effective data term captures the effects of the cross-lingual transfer:

$$D_x = D_{all}\left(1 + \gamma \sum_{L_i \neq L_j} p_{L_i} p_{L_j} \tau_{ij}\right) \quad (7)$$

where τ_{ij} is the transfer coefficient derived from Table 1.

Scaling Law under Optimal Allocation Substituting the optimal proportions $p*$ from Figure 5:

$$\mathcal{L}^*(N, D) = A^* \cdot N^{-\alpha_N^*} + B^* \cdot D^{-\alpha_D^*} + L_\infty^* \quad (8)$$

where $\alpha_D^* = 0.6859$, $\alpha_N^* = 0.2186$, $L_\infty^* = 0.2025$ are the fitted parameters under the optimal multilingual allocation for the multilingual code generation and translation at the same time. These coefficients are obtained through weighted fitting of the multilingual code generation and the translation loss.

Evaluation on Multilingual Code Generation and Translation Both LLMs are evaluated on MultiPL-E across all 7 PLs using Pass@1 and our code translation test set with a BLEU score. Figure 6 shows that optimized allocation outperforms uniform distribution under identical compute budgets. High-synergy pairs (e.g., JavaScript-TypeScript) benefit from balanced allocation, Python improves with increased data due to high α_D, while low-α_D languages (e.g., Rust) maintain performance despite fewer tokens. Importantly, no language suffers significant degradation, demonstrating that strategic reallocation achieves better equilibrium without creating imbalances. Our results provide concrete evidence that multilingual scaling laws can guide data allocation strategies.

7 Related Work

Scaling Laws The systematic study of scaling laws has evolved from early power-law observations (Hestness et al., 2019; Shallue et al., 2019) to foundational work (Kaplan et al., 2020) establishing predictable relationships between model size, dataset size, and compute. This was refined by (Hoffmann et al., 2022), demonstrating compute-optimal training requires equal scaling of parameters and training tokens. While the previous works (Wei et al., 2022) document emergent abilities that appear at certain scale thresholds and is later challenged as metric-dependent (Schaeffer et al., 2023). Recent work (Luo et al., 2025) shows code exhibits more data-hungry scaling than natural language.

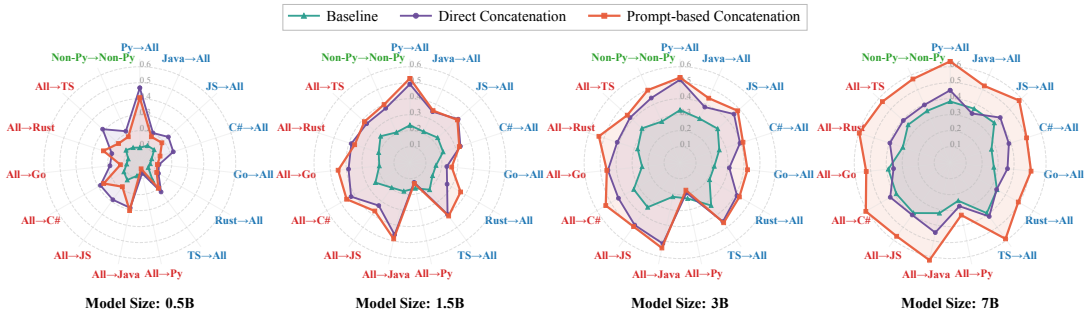

Figure 4: Translation scores across 3 strategies, 7 programming languages (PLs), and 42 directions. We aggregated the results by averaging based on language and direction into three categories: from each language to others, from others to a specific language, and between other languages, excluding Python. Across different model sizes, both **prompt-based concatenation** and **direct concatenation** significantly outperform the Baseline. Furthermore, we observe that scores for translations from other languages to Python are significantly lower than for other directions.

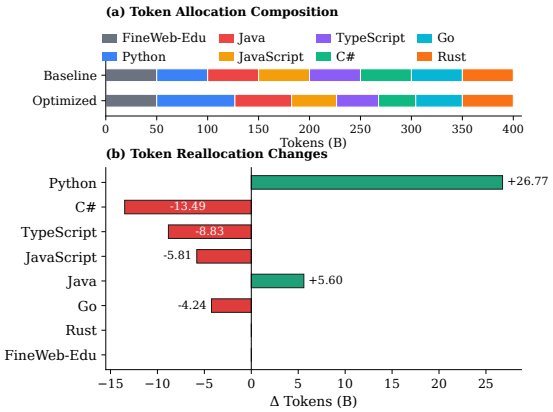

Figure 5: Token allocation comparison between baseline and optimized strategies. Both use 350B total code tokens but with different distributions. The optimized allocation is derived from fitted scaling laws (α_N, α_D, L_∞), optimal N:D ratios, and synergy gain analysis.

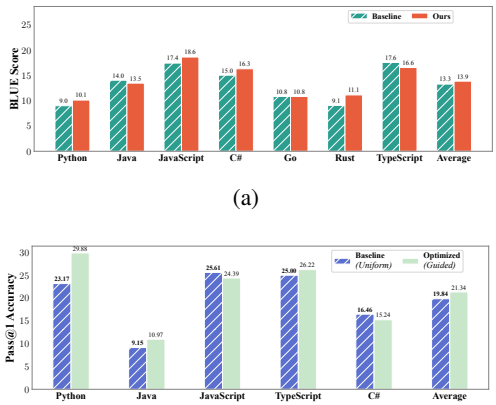

Figure 6: (a) The BLEU scores of the code translation and (b) Pass@1 accuracy on the MultiPL-E benchmark for baseline (uniform allocation) and optimized (guided allocation) strategies. Both LLMs are trained on 400B total tokens (350B code + 50B natural language text).

Code Pre-training Recent advancements in code large language models (code LLMs) have shown remarkable progress in understanding, generating, and reasoning about code (Guo et al., 2024; Hui et al., 2024; Li et al., 2023; Allal et al., 2023). Early works like CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021) applied masked language modeling and AST-based representations to source code. Subsequent models such as CodeT5 (Wang et al., 2021) and CodeGen (Nijkamp et al., 2023) adopted encoder-decoder architectures for generative tasks. Recent foundation models like Codex (Chen et al., 2021), StarCoder (Li et al., 2023), Code Llama (Roziere et al., 2023), QwenCoder (Hui et al., 2024), and DeepSeek-Coder (Guo et al., 2024) have scaled to trillions of tokens, employing pre-training and post-training (Le et al., 2022; Guo et al., 2025) to improve generalization and reasoning.

8 Conclusion

In this work, we present the first systematic investigation of scaling laws for multilingual code pre-training through 1000+ experiments, revealing that: (1) different PLs exhibit distinct scaling behaviors, with interpreted languages like Python showing larger scaling exponents than compiled languages like Rust; (2) strategic language pairing yields synergistic benefits, especially for syntactically similar languages; (3) parallel pairing strategies significantly enhance translation capabilities; and (4) the proportion-dependent multilingual scaling law enables optimal token allocation that improves average performance without degrading individual language capabilities, providing guidance for designing efficient multilingual code LLMs and establish a theoretical foundation for understanding cross-lingual transfer in code pre-training.

Limitations

This work presents the first comprehensive investigation of scaling laws for multilingual code pre-training, drawing insights from over 1000 experiments equivalent to 336,000+ H800 GPU hours across diverse programming languages, model architectures ranging from 0.2B to 14B parameters, and datasets up to 1T tokens. Our findings reveal that scaling behavior varies significantly across programming languages, with interpreted languages demonstrating greater performance gains from increased scale compared to compiled languages. We demonstrate that multilingual pre-training yields synergistic effects, particularly between linguistically similar programming languages, and that parallel code pairing substantially improves cross-lingual transfer capabilities. Based on these insights, we introduce a proportion-dependent scaling law that strategically allocates training tokens by emphasizing high-utility languages such as Python, optimizing synergistic language pairs like JavaScript-TypeScript, and reducing allocation to languages that saturate quickly such as Rust. This principled approach to token allocation consistently outperforms uniform distribution strategies, providing a practical framework for efficient multilingual code model pre-training that maximizes performance while optimizing computational resource utilization.

Ethics Statement

This research investigates scaling laws for multilingual code pre-training using publicly available code corpora and does not involve human subjects or personal data collection. The code translation evaluation set was annotated by professional software engineers who were fairly compensated and informed about the research purpose. We acknowledge that our extensive experiments (equivalent to 336,000+ H800 GPU hours) consume substantial computational resources and energy. However, by sharing our scaling laws and findings, we aim to reduce the need for future researchers to conduct similarly expensive experiments, thereby mitigating the overall environmental impact. We also recognize that code generation models trained on existing corpora may inherit biases present in programming practices and documentation styles. We encourage responsible deployment of code generation technologies with appropriate safeguards to prevent potential misuse.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: Don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Anysphere Inc. 2025. Cursor features. https://cursor.com/features.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, abs/2107.03374.

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, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. 2019. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Xueyu Hu, Tao Xiong, Biao Yi, Zishu Wei, Ruixuan Xiao, Yurun Chen, Jiasheng Ye, Meiling Tao, Xiangxin Zhou, Ziyu Zhao, et al. 2025. Os agents: A survey on mllm-based agents for general computing devices use. *arXiv preprint arXiv:2508.04482*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

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 preprint arXiv:2001.08361*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, abs/2305.06161.

Xianzhen Luo, Wenzhen Zheng, Qingfu Zhu, Rongyi Zhang, Houyi Li, Siming Huang, YuanTao Fan, and Wanxiang Che. 2025. Scaling laws for code: A more data-hungry regime. *arXiv preprint arXiv:2510.08702*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code.

Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. 2023. Are emergent abilities of large language models a mirage? *Advances in neural information processing systems*, 36:55565–55581.

Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. 2019. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, 20(112):1–49.

Shuai Wang, Weiwen Liu, Jingxuan Chen, Yuqi Zhou, Weinan Gan, Xingshan Zeng, Yuhan Che, Shuai Yu, Xinlong Hao, Kun Shao, et al. 2024. Gui agents with foundation models: A comprehensive survey. *arXiv preprint arXiv:2411.04890*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.