Scaling Laws for Code: Every Programming Language Matters

Jian Yang[1] Shawn Guo[2] Lin Jing[2] Wei Zhang[1] Aishan Liu[1] Chuan Hao[2]
Zhoujun Li[1] Wayne Xin Zhao[3] Xianglong Liu[1†] Weifeng Lv[1†] Bryan Dai[2]
[1]Beihang University [2]Ubiquant
[3]Gaoling School of Artificial Intelligence, Renmin University of China
[†]Corresponding Authors. Email: {xlliu, lwf}@buaa.edu.cn

Abstract

Code large language models (Code 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. Besides, 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 comprehensive scaling laws for code LLMs across multiple PLs, revealing that interpreted languages (e.g., Python) benefit more from increased model size and data than compiled languages (e.g., Rust). The study demonstrates that multilingual pre-training provides synergistic benefits, particularly between syntactically similar PLs. Further, the pre-training strategy of the parallel pairing (concatenating code snippets with their translations) significantly enhances cross-lingual abilities with favorable scaling properties. Finally, a proportion-dependent multilingual scaling law is proposed to optimally allocate training tokens by prioritizing high-utility PLs (e.g., Python), balancing high-synergy pairs (e.g., JavaScript-TypeScript), and reducing allocation to fast-saturating languages (Rust), achieving superior average performance across all PLs compared to uniform distribution under the same compute budget.

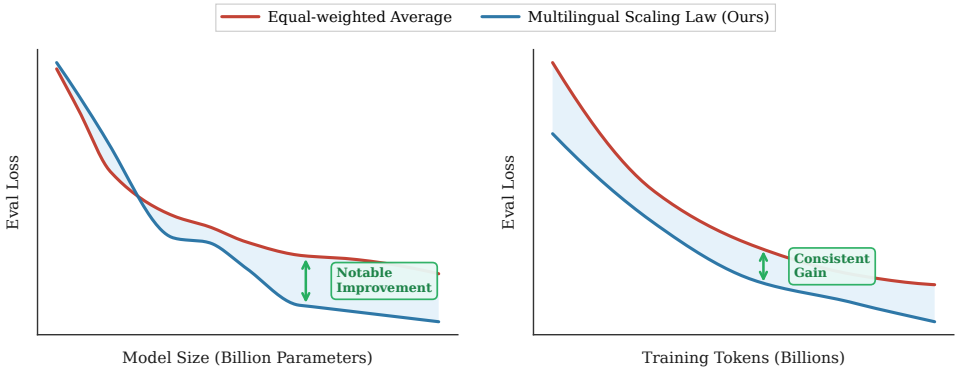


Figure 1. Evaluation loss comparison showing that the proposed multilingual scaling law achieves consistently lower loss than the baseline across model sizes and token budgets.

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 [3, 5, 8, 13, 14, 25]. Code LLMs significantly enhance developer productivity [2], but training top-tier LLMs consumes enormous computing resources and costs [3, 4, 11, 16], making it prohibitively expensive to conduct ablation experiments on data composition (e.g., proportions of different PLs) 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.

Scaling laws in general domains present a principled approach to characterizing how model performance (validation loss) depends on essential variables, such as model size, dataset size, and compute budget [11, 16]. The recent work [19] has extended scaling laws from natural language to code, revealing that code pre-training is significantly more data-hungry compared to standard pre-training, which requires larger data-to-parameter ratios to achieve optimal performance. Yet modern code pre-training software development is inherently multilingual, with developers routinely working across Python, Java, JavaScript, C++, and numerous other PLs for downstream applications and tasks (e.g., OSAgents [12] and TerminalBench [24]). This reality raises critical unanswered questions: *(1) What is the scaling law for multilingual code pre-training and multilingual cross-lingual capabilities? (2) What is the optimal strategy for allocating training resources across different PLs?*

Understanding multilingual scaling dynamics is crucial for several reasons. First, real-world code repositories exhibit substantial linguistic diversity, with languages varying in syntactic structure, semantic properties, and data availability. Second, multilingual pretraining may enable cross-lingual transfer, where knowledge from one language improves performance on others. Third, programming languages universally share core semantic concepts for manipulating data and controlling logic, even though their surface-level syntax and expressive paradigms can differ dramatically. Despite the practical importance of these questions, existing scaling law research focuses exclusively on monolingual or language-agnostic settings, leaving the multilingual dimension unexplored.

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 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:

- We establish language-specific scaling laws for each PLs independently, revealing that interpreted PLs (e.g., Python) show larger scaling exponents (benefiting more from increased model size and data) compared to compiled PLs (e.g., Rust). The irreducible loss metric establishes a complexity ordering (C# < Java ≈ Rust < Go < TypeScript < JavaScript < Python), where PLs with strict syntax are more learnable and predictable than dynamically-typed PLs.

- We study synergy gain matrix of different PLs, showing that PLs with similar syntax or structure (e.g., Java-C#) can bring positive transfer compared to single-PL pre-training. Most PLs can benefit from multilingual pre-training, indicating that optimal PL mixing strategies in pre-training must be tailored to the characteristics of each PL.
- This research investigates how data organization strategies during pre-training affect cross-lingual abilities of code LLMs. The experiments discover that parallel pairing, which concatenates code snippets with their translations to provide explicit document-level alignment, significantly outperforms baseline on multilingual code translation and generation. The parallel pairing strategy exhibits favorable scaling laws, with a high scaling exponent, enabling efficient utilization of model capacity to learn cross-lingual alignments of different translation directions.
- We propose a proportion-dependent multilingual scaling law that incorporates language-specific parameters $(\alpha_N, \alpha_D, L_\infty)$ and the parameter of cross-lingual transfer effects τ_{ij}. Under optimal allocation, the model allocates more tokens to high-utility languages (e.g., Python) and balanced amounts to high-synergy pairs (e.g., JavaScript-TypeScript) while reducing tokens for fast-saturating languages (e.g., Rust). Evaluation of the multilingual code generation demonstrates that optimized allocation achieves higher average performance across all PLs without significant degradation in any single language, validating that strategic reallocation based on scaling laws and language synergies outperforms uniform distribution under identical compute budgets.

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 [11] as below:

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

where $\mathcal{L}$ is the cross-entropy loss, 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 irreducible loss represents the inherent error that no model can eliminate, regardless of its size or training quality.).

2.2. Motivation and Formulation

Our goal is to address three fundamental questions about multilingual code pre-training through a systematic empirical investigation:

(1) Language-Specific Scaling Dynamics: How does each PL scale with model size and data? We aim to investigate whether different languages exhibit distinct scaling exponents and irreducible losses, thereby challenging the assumption that all programming languages can be treated uniformly in pre-training.

(2) Cross-Lingual Synergy Effects: What are the synergistic or antagonistic effects when mixing different PLs during pre-training? We investigate whether bilingual training can out-

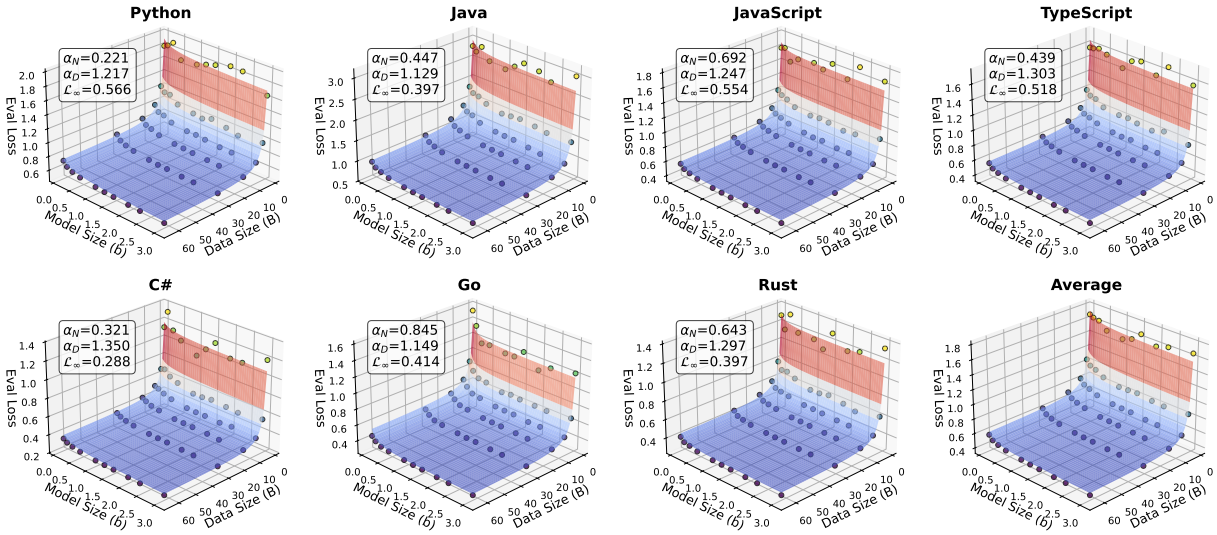


Figure 2. Scaling Laws for each PL independently. It shows a clear ordering of intrinsic predictability across PLs Under the same data scale: C# < Java ≈ Rust < Go < TypeScript < JavaScript < Python.

perform monolingual baselines under fixed compute budgets and identify which language combinations yield positive transfer versus harmful interference.

(3) Compositional Cross-Lingual Transfer: Can LLMs trained on supervised language pairs generalize to unseen language pairs? We examine whether different data organization strategies (from random shuffling to supervised pair training) enable zero-shot transfer between languages without direct parallel data.

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

TAKEAWAYS: Language-specifc Scaling Law

- For code across all PLs, scaling up data size yields greater performance gains than scaling up model size.
- Different PLs exhibit distinct convergence rates during training.
- PLs 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) × 6 (token budgets) × 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 systematic variations in model size (N) and training tokens (D). For each language, we fit both the Chinchilla-style power law.

Scaling Exponents and Optimal Allocation Figure 2 summarizes the fitted scaling parameters for each PL. The results reveal significant heterogeneity in scaling behaviors across PLs, challenging the assumption that all programming languages can be treated uniformly. The results show a clear pattern that interpreted languages exhibit larger scaling exponents than compiled languages. Among dynamically-typed interpreted PLs, Python obtains the highest α_N and α_D values, indicating that it benefits more aggressively from increases in both model parameters and training data. In contrast, Rust, as a statically-typed compiled PL with strict memory safety guarantees, shows notably smaller exponents. The explicit type annotations and rigid syntactic structure of Rust make it more learnable for LLMs with fewer parameters and less training data compared to dynamically-typed languages. The optimal N:D ratios also vary substantially across languages, reflecting different compute-data trade-offs. PLs with higher α_D relative to α_N (e.g., Python) favor larger training datasets for a given parameter budget, while PLs with lower α_D (e.g., Rust) can achieve comparable performance with relatively fewer tokens but may benefit from increased model capacity.

Irreducible Loss and Language Complexity The irreducible loss $\mathcal{L}_\infty$ serves as a fundamental measure of language complexity, where the lower bound on achievable perplexity given infinite compute. Our results show a clear ordering: C# < Java ≈ Rust < Go < TypeScript < JavaScript < Python, providing insights into the intrinsic predictability of different programming languages. C# achieves the lowest $\mathcal{L}_\infty$ due to its strict type system, consistent naming conventions, and standardized ecosystem. Java and Rust both enforce strong syntactic and semantic constraints that limit expression diversity. The minimalist design of Go3 yields moderate predictability, while TypeScript retains partial unpredictability from JavaScript through optional typing. The high $\mathcal{L}_\infty$ of JavaScript arises from its dynamic typing, flexible paradigms, and lack of uniform standards. Python exhibits the highest $\mathcal{L}_\infty$, reflecting its dynamic and expressive nature, diverse idioms, and variability in coding style.

4. Language Mixture for Data Scarcity

TAKEAWAYS: Language-specifc Scaling Law

- For code data across all programming languages, scaling up data size yields greater performance gains than scaling up model size.
- 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 ($|D_{L_i}|$ = 64B) and PL L_j ($|D_{L_j}|$ = 64B) 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:

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

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 **??** presents the synergy gain $\Delta(\mathcal{L}_i, \mathcal{L}_j)$ for all language pairs. A positive value indicates that PL L_i mixing with the auxiliary PL L_j improves performance on the target PL L_i compared to the self-repetition baseline ($D_{L_i} \times 2$). The diagonal entries are zero by definition (self-repetition baseline). Our results reveal two key findings: (1) The multilingual pre-training provides substantial benefits for most PLs, with 6/7 languages showing consistent positive synergy across all auxiliary language combinations. (2) PLs that share similar syntax, semantics, or programming paradigms benefit significantly from joint training, as the LLM can leverage shared statistical patterns and transfer learned representations across languages. Java exhibits exceptional synergy gains with all auxiliary PLs. Mixing Java with C# yields the largest improvement (Δ = 0.186), followed by JavaScript (Δ = 0.114), TypeScript (Δ = 0.109), and Rust (Δ = 0.112). This suggests that Java, despite its mature ecosystem, benefits enormously from exposure to diverse programming paradigms during pre-training. The Java-C# pair likely benefits from their shared object-oriented paradigm, similar standard libraries, and common design patterns.

Target	Python	Java	JavaScript	TypeScript	C#	Go	Rust
Python	0.7528	0.7426 (↑1%)	0.7613 (↓1%)	0.7600 (↓1%)	0.7656 (↓2%)	0.7688 (↓2%)	0.7733 (↓3%)
Java	0.8490 (↑6%)	0.9034	0.7894 (↑13%)	0.7942 (↑12%)	0.7175 (↑21%)	0.8069 (↑11%)	0.7913 (↑12%)
JavaScript	0.5126 (↑5%)	0.5262 (↑3%)	0.5424	0.5170 (↑5%)	0.5292 (↑2%)	0.5352 (↑1%)	0.5285 (↑3%)
TypeScript	0.5124 (↑4%)	0.5225 (↑2%)	0.5169 (↑3%)	0.5347	0.5257 (↑2%)	0.5273 (↑1%)	0.5284 (↑1%)
C#	0.3327 (↑4%)	0.3393 (↑2%)	0.3352 (↑3%)	0.3331 (↑4%)	0.3459	0.3395 (↑2%)	0.3391 (↑2%)
Go	0.4121 (↑5%)	0.4200 (↑3%)	0.4211 (↑3%)	0.4137 (↑4%)	0.4200 (↑3%)	0.4328	0.4204 (↑3%)
Rust	0.3801 (↑4%)	0.3840 (↑3%)	0.3788 (↑4%)	0.3794 (↑4%)	0.3843 (↑3%)	0.3844 (↑3%)	0.3954

Table 1. Synergy gain matrix. Values indicate absolute performance (pass@1), parentheses show relative change vs. diagonal baseline. **Red**: performance gain; **Teal**: performance decline. Background intensity indicates magnitude.

We also observe that certain language pairs outperform self-repetition baselines. The Java-C# combination achieves validation loss of 0.718 compared to 0.903 for Java self-repetition—a remarkable 20.5% improvement. Similarly, JavaScript-TypeScript mixing (0.517 vs 0.542) and TypeScript-JavaScript mixing (0.517 vs 0.535) both outperform their respective self-repetition baselines. This suggests that exposure to a related but distinct language introduces beneficial regularization or diversifies the learned representations in a way that substantially improves generalization.

However, when Python is the target PL, mixing with most auxiliary PL produces small negative effects: JavaScript ($\Delta = -0.009$), TypeScript ($\Delta = -0.007$), C# ($\Delta = -0.013$), Go ($\Delta = -0.016$), and Rust ($\Delta = -0.021$). Only Java provides a modest positive synergy ($\Delta = 0.010$). The cross-lingual transfer is asymmetric: mixing Python as an auxiliary language benefits other target languages (e.g., $\Delta = 0.054$ for Java, $\Delta = 0.030$ for JavaScript), but Python itself suffers when mixed with them. Overall, negative interference is limited and language-specific rather than a general phenomenon. The results indicate that multilingual pre-training benefits most programming languages, 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, prioritizing mixing syntactically-related PLs can further bring more improvement compared to naively upsampling a single PL. The positive synergy effects suggest that linguistic diversity, particularly when it spans across the code domain, acts as a form of 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 the experiments of this section use a 50:50 ratio for language mixing, section 6 will investigate whether asymmetric mixtures (e.g., 75:25) yield further improvements.

5. Cross-Lingual Scaling Laws

TAKEAWAYS: 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#`).

5.1. Experimental Setting

Pre-training Strategies This section investigates how different pre-training strategies affect cross-lingual transfer capabilities. Given K PLs, there are $n = K \times (K-1)$ translation directions in total. We only have Python-centric training parallel data ($t = 2K$ translation directions), while the other $m = n - t$ directions have no data. (1) The LLM is pre-trained on $\{D_1, \ldots, D_K\}$ (D_k denotes the dataset of PL L_k), and then we evaluate the average translation loss of different PLs on $\{L_{i_1 \to j_1}, \ldots, L_{i_t \to j_t}\}$ ($L_{i_1 \to j_1}$ denotes the translation direction from PL L_{i_1} to L_{j_1}) (2) The LLM is pre-trained on t translation direction $\{D_{i_1 \to j_1}, \ldots, D_{i_t \to j_t}\}$, and then we evaluate the average translation loss on t translation direction $\{L_{i_1 \to j_1}, \ldots, L_{i_t \to j_t}\}$. (3) The LLM is pre-trained on Python-centric parallel multilingual data $\{D_{L_{i_1 \to j_1}}, \ldots, D_{i_t \to j_t}\}$, and then we evaluate the average translation loss on m unseen translation directions $\{L_{i_{t+1} \to j_{t+1}}, \ldots, L_{i_n \to j_n}\}$.

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 To assess cross-lingual capabilities, we construct a held-out evaluation set for PL translation task through a careful curation process. 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 Multilingual 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:

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

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, larger LLMs develop emergent cross-lingual capability, suggesting that pre-training on unsupervised multilingual code data can gets the basic cross-lingual capability.

	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: $\mathcal{L}(N, D) = A/N^{\alpha_N} + B/D^{\alpha_D} + L_\infty$.

5.3. Pre-training on Supervised Multilingual 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 ↔ {Java, JavaScript, TypeScript, C#, Go, Rust}), we observe enhanced cross-lingual performance that scales as:

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

where $A_a = 0.0508$,$B_a = 0.793$, $\alpha_a = 6.404$,$\beta_a = 0.8829$, and $L_{\infty,a} = 0.1006$. where 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.

Zero-shot Translation Scaling Law Critically, 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:

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

where $A_{zt} = 0.0350$, $B_{zt} = 4.518$, $\alpha_{zt} = 0.781$, $\beta_{zt} = 0.869$, and $L_{\infty,zt} = 0.0524$. The equation reveal that zero-shot performance under document-level pairing substantially exceeds that of the shuffled baseline. This suggests that the model leverages Python as an implicit bridge, composing translations through learned bidirectional mappings (e.g., Java → Python → Go). Table 2 summarizes the fitted scaling parameters across strategies and evaluation settings, demonstrating that parallel pairing yields superior scaling performance for both seen and unseen translation diretions.

5.4. Cross-Lingual Translation Strategies

This subsection examines 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

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 VS Doc Level

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 ($x + y$).

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.

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.

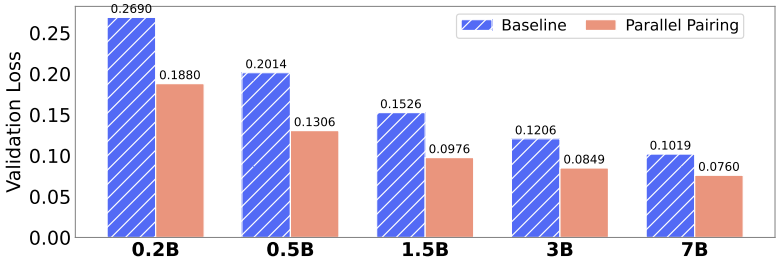


Figure 3. Validation loss on unseen translation directions (non-Python language pairs). Each entry is the average loss across 30 translation pairs not seen during pre-training.

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.

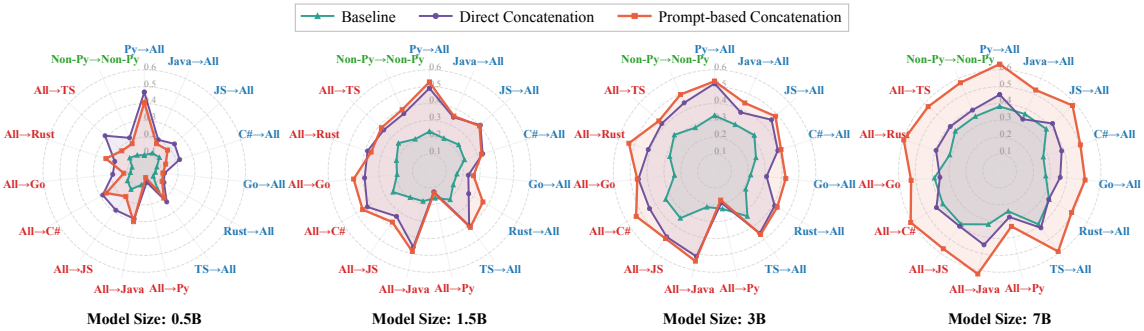


Figure 4. Translation scores across 3 strategies, 7 programming languages (PLs), and 42 directions. We aggregated the results by averaging based on language and direction into three categories: from each language to others, from others to a specific language, and between other languages, excluding Python. 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.

6. Guideline for Code Pre-training

TAKEAWAYS: Cross-Lingual Pre-training Strategies

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

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

(3) The methodology can generalize beyond seven codes to any multilingual code pre-training setting.

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 languages (Python), balanced allocation for high-synergy pairs (Java-C#, JavaScript-TypeScript), and reduced tokens for fast-saturating languages (e.g, Go).

6.2. 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:

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

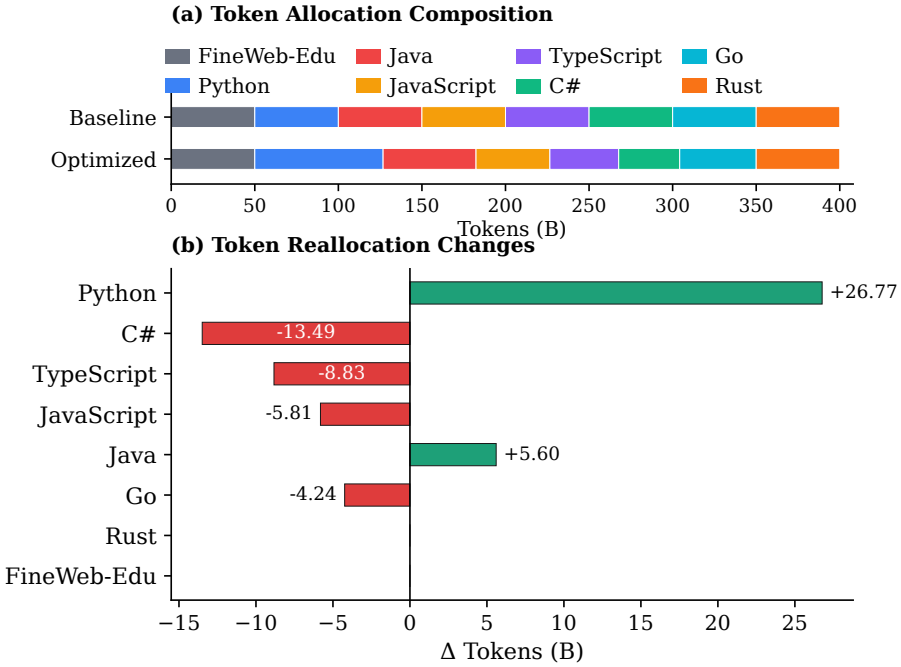


Figure 5. Token allocation comparison between baseline and optimized strategies. Both use 350B total code tokens but with different distributions. The optimized allocation is derived from fitted scaling laws (α_N, α_D, L_∞), optimal N:D ratios, and synergy gain analysis.

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:

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

where τ_{ij} is the transfer coefficient derived from **??**.

6.3. Scaling Law under Optimal Allocation

Substituting the optimal proportions $p*$ from Figure 5:

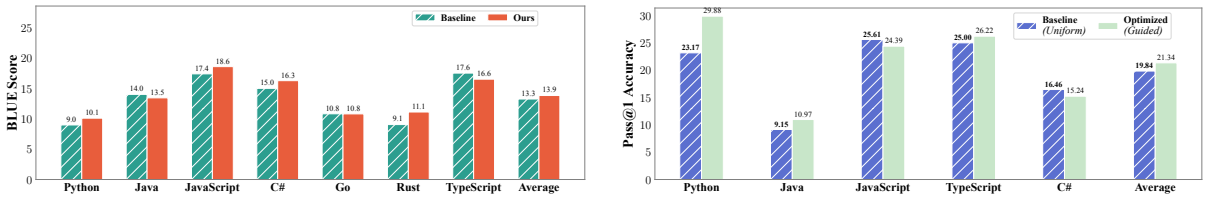
$$\mathcal{L}^*(N, D) = A^* \cdot N^{-\alpha_N^*} + B^* \cdot D^{-\alpha_D^*} + L_\infty^* \tag{8}$$

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.

6.4. Evaluation on Multilingual Code Generation and Translation

Both LLMs are evaluated on the multilingual code generation benchmark MultiPL-E across all 7 PLs with the Pass@1 metric and our created code translation test set with the BLEU score. Figure 6 validates that the optimized allocation achieves higher average performance than uniform distribution under identical compute budgets. The results show that high-synergy

pairs (e.g., JavaScript-TypeScript) benefit most from balanced allocation, Python improves with increased data due to high α_D, while low-α_D languages (e.g., Rust) maintain strong performance despite reduced tokens. Importantly, no language suffers significant degradation, demonstrating that strategic reallocation finds a better equilibrium without creating imbalances. Our results provide concrete evidence that multilingual scaling laws can directly inform data allocation strategies for each PL.



(a) Multilingual translation performance

(b) Multilingual code generation performance

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

7. Related Work

Scaling Laws The systematic study of scaling laws in neural networks has evolved significantly since early observations [10, 23] show power-law relationships between scale and performance. The foundational work [16] established that language model performance scales predictably with model size, dataset size, and compute, though this was significantly [11] in the Chinchilla paper, which demonstrated that models should be scaled equally in parameters and training tokens for compute-optimal training. This field has expanded to cover various domains, including transfer learning, data quality, and generative modeling beyond language, while the previous works [27] documented emergent abilities that appear at certain scale thresholds, which is a phenomenon later challenged by potentially metric-dependent [22]. The pioneering study about code scaling [19] establishes that code exhibits distinct and more data-hungry scaling laws than natural language, requiring a higher optimal data-to-parameter ratio but limited to the analysis of a single programming language.

Code Pre-training Recent advancements in code large language model (code LLM) pre-training have demonstrated remarkable progress in enabling models to understand, generate, and reason about programming languages [1, 8, 13, 18]. Early works, such as CodeBERT [6] and GraphCodeBERT [7], extend natural language pre-training techniques like masked language modeling and replaced token detection to source code, integrating both textual and structural representations through abstract syntax trees (ASTs). Subsequent efforts, including CodeT5 [26] and CodeGen [20], adopt encoder-decoder architectures for bidirectional and generative code tasks, leveraging large-scale datasets such as CodeSearchNet [15]. More recent foundation models like Codex [5], StarCoder [18], Code Llama [21], QwenCoder [13], and DeepSeek-Coder [8] have scaled pre-training to trillions of tokens across multiple programming languages, employing pre-training and post-training [9, 17] to improve generalization, reasoning, and alignment with human intent.

8. Conclusion

In this work, we present the first systematic investigation of scaling laws for multilingual code pre-training, addressing critical gaps in our understanding of how programming language diversity affects model performance. Through over 1000+ experiments spanning multiple languages, model sizes, and dataset configurations, we establish several key findings: (1) different programming languages exhibit distinct scaling behaviors, with interpreted languages like Python showing larger scaling exponents than compiled languages like Rust, and intrinsic complexity ordering from C# to Python reflected in their irreducible losses; (2) strategic language pairing during pre-training yields synergistic benefits, particularly for syntactically similar languages like Java-C#, while most languages benefit from multilingual exposure compared to monolingual training; (3) parallel pairing strategies for organizing cross-lingual data significantly enhance translation capabilities with favorable scaling properties; and (4) our proportion-dependent multilingual scaling law enables optimal token allocation strategies that allocate more resources to high-utility languages and synergistic pairs while reducing allocation to fast-saturating languages, achieving superior average performance without degrading individual language capabilities. These findings provide actionable guidance for practitioners designing multilingual code models and establish a theoretical foundation for understanding cross-lingual transfer in code pre-training, ultimately enabling more efficient utilization of computational resources in training next-generation code LLMs.

Limitations

While our work provides comprehensive insights into multilingual code scaling laws, several limitations warrant consideration. First, our experiments cover only seven programming languages, which, although representative of diverse paradigms, constitute a subset of production languages; extending findings to low-resource or domain-specific languages (e.g., SQL, assembly) remains unexplored. Second, our largest model reaches 14B parameters with 1T tokens, whereas state-of-the-art code LLMs exceed 100B parameters trained on multiple trillions of tokens (whether our scaling exponents hold at extreme scales requires validation). Third, our evaluation focuses on code translation and generation benchmarks, which may not fully capture performance on complex tasks like program repair or multi-file completion. Fourth, the synergy coefficients are fitted to our specific corpus; different data distributions may yield varying patterns requiring recalibration. Finally, our optimization assumes fixed token budgets and does not explore dynamic curriculum learning or adaptive sampling strategies that could further improve multilingual performance.

Acknowledgment

This work was supported by State Key Laboratory of Complex & Critical Software Environment (SKLCCSE) of Beihang University and supported by the Fundamental Research Funds for the Central Universities. This work was supported in part by the National Natural Science Foundation of China (Grant Nos. 62276017, 62406033, U1636211, 61672081), and the State Key Laboratory of Complex & Critical Software Environment (Grant No. SKLCCSE-2024ZX-18).

References

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

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

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

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

5 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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

6 Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 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, 2020. doi: 10.18653/V1/2020.FINDINGS-EMNLP.139. URL https://doi.org/10.18653/v1/2020.findings-emnlp.139.

7 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. 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, 2021. URL https://openreview.net/forum?id=jLoC4ez43PZ.

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

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

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

11 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. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

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

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

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

15 Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, abs/1909.09436, 2019. URL http://arxiv.org/abs/1909.09436.

16 Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

17 Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *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*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html.

18 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. StarCoder: May the source be with you! *arXiv preprint*

arXiv:2305.06161, abs/2305.06161, 2023. doi: 10.48550/arXiv.2305.06161. URL https://doi.org/10.48550/arXiv.2305.06161.

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

20 Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 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, 2023.

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

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

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

24 The Terminal-Bench Team. Terminal-bench: A benchmark for ai agents in terminal environments, Apr 2025. URL https://github.com/laude-institute/terminal-bench.

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

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

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