Granite Code Models: A Family of Open Foundation Models for Code Intelligence

Mayank Mishra* **Matt Stallone*** **Gaoyuan Zhang*** **Yikang Shen** **Aditya Prasad**
Adriana Meza Soria **Michele Merler** **Parameswaran Selvam** **Saptha Surendran**
Shivdeep Singh **Manish Sethi** **Xuan-Hong Dang** **Pengyuan Li** **Kun-Lung Wu**
Syed Zawad **Andrew Coleman** **Matthew White** **Mark Lewis** **Raju Pavuluri**
Yan Koyfman **Boris Lublinsky** **Maximilien de Bayser** **Ibrahim Abdelaziz**
Kinjal Basu **Mayank Agarwal** **Yi Zhou** **Chris Johnson** **Aanchal Goyal** **Hima Patel**
Yousaf Shah **Petros Zerfos** **Heiko Ludwig** **Asim Munawar** **Maxwell Crouse**
Pavan Kapanipathi **Shweta Salaria** **Bob Calio** **Sophia Wen** **Seetharami Seelam**
Brian Belgodere **Carlos Fonseca** **Amith Singhee** **Nirmit Desai** **David D. Cox**
Ruchir Puri† **Rameswar Panda**†

IBM Research
*Equal Contribution
†Corresponding Authors
ruchir@us.ibm.com, rpanda@ibm.com

Abstract

Large Language Models (LLMs) trained on code are revolutionizing the software development process. Increasingly, code LLMs are being integrated into software development environments to improve the productivity of human programmers, and LLM-based agents are beginning to show promise for handling complex tasks autonomously. Realizing the full potential of code LLMs requires a wide range of capabilities, including code generation, fixing bugs, explaining and documenting code, maintaining repositories, and more. In this work, we introduce the Granite series of decoder-only code models for code generative tasks, trained with code written in 116 programming languages. The Granite Code models family consists of models ranging in size from 3 to 34 billion parameters, suitable for applications ranging from complex application modernization tasks to on-device memory-constrained use cases. Evaluation on a comprehensive set of tasks demonstrates that Granite Code models consistently reaches state-of-the-art performance among available open-source code LLMs. The Granite Code model family was optimized for enterprise software development workflows and performs well across a range of coding tasks (e.g. code generation, fixing and explanation), making it a versatile "all around" code model. We release all our Granite Code models under an Apache 2.0 license for both research and commercial use.

REPO: https://github.com/ibm-granite/granite-code-models

1 Introduction

Over the last several decades, software has been woven into the fabric of every aspect of our society. As demand for software development surges, it is more critical than ever to increase software development productivity, and LLMs provide promising path for augmenting human programmers. Prominent enterprise use cases for LLMs in software development productivity include code generation, code explanation, code fixing, unit test and documentation generation, application modernization, vulnerability detection, code translation, and more.

Recent years have seen rapid progress in LLM's ability to generate and manipulate code, and a range of models with impressive coding abilities are available today. Models range in

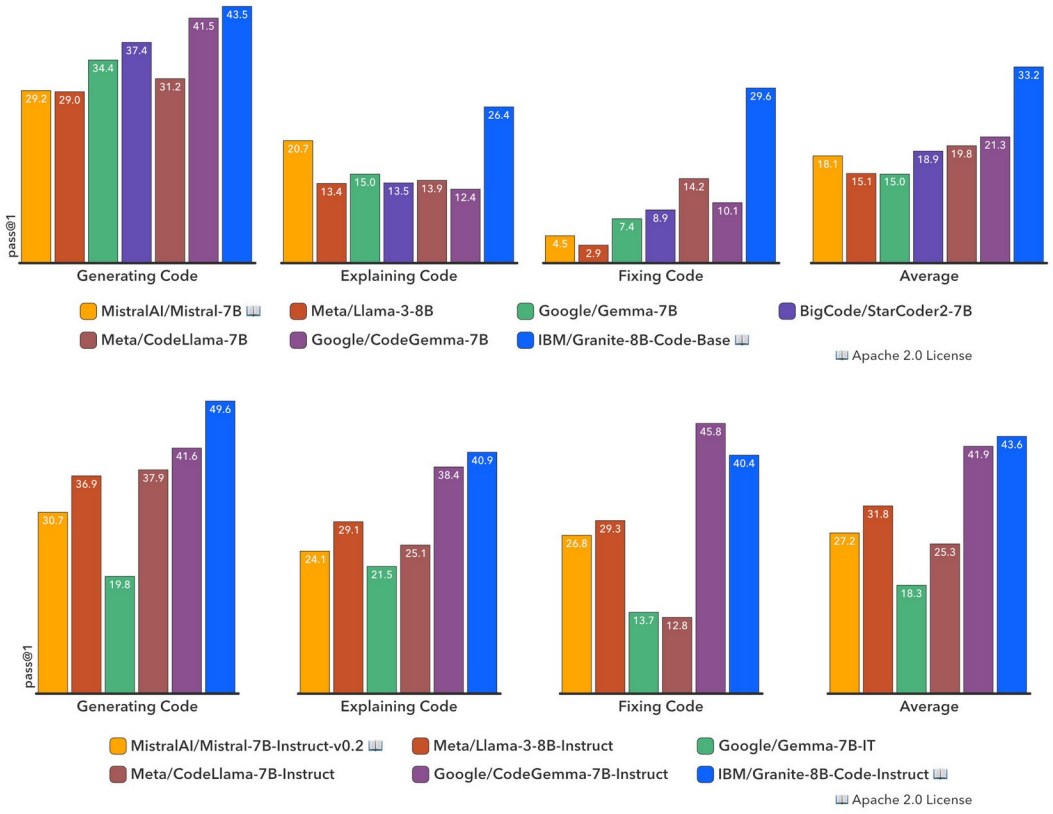

Figure 1: Comparison of Granite-8B-Code (Base/Instruct) with other open source (code) LLMs of similar size on HumanEvalPack (Muennighoff et al., 2023), spanning 3 coding tasks and 6 programming languages. See Tables 3,10,11 for more details. Best viewed in color.

size from single-digit billions of parameters (e.g. Llama-7B (Touvron et al., 2023), Gemma-7B (Gemma-Team et al., 2024), etc.) to hundreds of billions: DBRX (Databricks), Arctic (Snowflake), Grok, Mixtral 8x22B (MistralAI), Command R+ (Cohere), and vary in the generality of intended use, with some models aiming to cover a range of uses outside of code, while others focus primarily on coding-related tasks (e.g. StarCoder (Li et al., 2023a; Lozhkov et al., 2024), CodeGen (Nijkamp et al., 2023), CodeLlama (Rozi ère et al., 2023), and CodeGemma (CodeGemma Team et al., 2024)).

However, there remain important gaps in the current field of LLMs for code, especially in the context of enterprise software development. First, while very large, generalist LLMs can achieve excellent coding performance, their size makes them expensive to deploy. Smaller code-focused models (Li et al., 2023a; Lozhkov et al., 2024; Nijkamp et al., 2023; Rozère et al., 2023; CodeGemma Team et al., 2024) can achieve excellent code generation performance in a smaller and more flexible package, but performance in coding tasks beyond generation (e.g. fixing and explanation) can lag behind code generation performance.

In many enterprise contexts, code LLM adoption can be further complicated by factors beyond the performance of the models. For instance, even open models are sometimes plagued by a lack of transparency about the data sources and data processing methods that went into model, which can erode trust in models in mission critical and regulated contexts. Furthermore, license terms in today's open LLMs can encumber and complicate an enterprise's ability to use a model.

Here, we present Granite Code models, a series of highly capable code LLMs, designed to support enterprise software development across a wide range of coding tasks. Granite Code models has two main variants that we release in four different sizes (3B, 8B, 20B, and 34B):

- **Granite Code Base:** base foundation models for code-related tasks;
- **Granite Code Instruct:** instruction following models finetuned using a combination of Git commits paired with human instructions and open-source synthetically generated code instruction datasets.

The base models in the series have been trained from scratch with a two-phase training strategy. In phase 1, our model is trained on 3 to 4 trillion tokens sourced from 116 programming languages, ensuring a comprehensive understanding of programming languages and syntax. In phase 2, our model is further trained on 500 billion tokens with a carefully designed mixture of high-quality data from code and natural language domains to improve the model's ability to reason. We use the unsupervised language modeling objective to train the base models in both the phases of training. The instruct models are derived by further finetuning the above trained base models on a combination of a filtered variant of CommitPack (Muennighoff et al., 2023), natural language instruction following datasets (OASST (Köpf et al., 2023), HelpSteer (Wang et al., 2023)) and open-source math datasets (MathInstruct (Yue et al., 2023) and MetaMathQA (Yu et al., 2023)), including synthetically generated code datasets for improving instruction following and reasoning capabilities.

We conduct extensive evaluations of our code LLMs on a comprehensive set of benchmarks, including HumanEvalPack (Muennighoff et al., 2023), MBPP(+) (Austin et al., 2021; Liu et al., 2023a), RepoBench (Liu et al., 2023b), ReCode (Wang et al., 2022), and more. This set of benchmarks encompasses many different kinds of coding tasks beyond just code synthesis in Python, e.g., code fixing, code explanation, code editing, code translation, etc., across most major programming languages (Python, JavaScript, Java, Go, C++, Rust, etc.).

Our findings reveal that among open-source models, the Granite Code models overall show very strong performance across all model sizes and benchmarks (often outperforming other open-source code models that are twice large compared to Granite). As an illustration, figure 1 (top) shows a comparison of Granite-8B-Code-Base with other open-source base code LLMs, including recent high-performing general purpose base LLMs like Mistral (Jiang et al., 2023b) and LLama-3 (AI@Meta, 2024) on HumanEvalPack (Muennighoff et al., 2023). While CodeGemma and StarCoder2 perform reasonably well in generating code, they perform significantly worse on the code fixing and explanation variants of HumanEvalPack. On average, Granite-8B-Code-Base outperforms the most competitive CodeGemma-8B model by almost 12 points on HumanEvalPack (33.2% vs 21.3%), despite being trained on significantly less number of tokens (4.5T vs 7.5T tokens). Besides base models, the instruction tuned variants of our Granite Code models also show strong performance on HumanEvalPack, outperforming other open-source (code) instruction models, demonstrating benefits to a wider set of coding tasks with natural language instructions (see figure 1 (bottom)).

Furthermore, since reasoning is critical for solving complicated questions and tasks, we also test our Granite-8B-Code-Base model on six mathematical benchmarks, including MATH (Cobbe et al., 2021), GSM8K (Cobbe et al., 2021) and problem solving with access to computational tools, where our Granite 8B model achieves better performance compared to most state-of-the-art 7B or 8B LLMs. For example, Granite-8B-Code-Base outperforms Llama-3-8B-Base by ∼12 points on GSM8K and ∼6 points on MATH (see table 15).

The key advantages of Granite Code models include:

- **All-rounder Code LLM**: Granite Code models achieve competitive or state-of-the-art performance on different kinds of code-related tasks, including code generation, explanation, fixing, editing, translation, etc., demonstrating their ability to solve diverse coding tasks;
- **Trustworthy Enterprise-Grade LLM**: All our models are trained on license-permissible data collected following IBM's AI Ethics principles [1] and guided by IBM's Corporate Legal team for trustworthy enterprise usage. All the Granite Code models are released under the Apache 2.0 license.

[1] https://www.ibm.com/impact/ai-ethics

We describe our entire data collection, filtering, and preprocessing pipeline in section 2. Section 3 describes the details of model architecture, followed by training details in Section 4. Section 5 provides the details about instruction tuning, and Section 6 describes the experiments and results comparing Granite Code models with other open-source LLMs.

2 Data Collection

In this section, we describe the process of crawling and filtering (Sec. 2.1), deduplication (Sec. 2.2), HAP/PII filtering (Sec. 2.3) used to prepare the code data for model training. We also provide an overview of high-quality natural language data used to enhance the model's language understanding and mathematical reasoning skills.

2.1 Data Crawling and Filtering

The pretraining code data was sourced from a combination of publicly available datasets like Github Code Clean[2], StarCoderdata[3], and additional public code repositories and issues from GitHub. We filter raw data to retain a list of 116 programming languages out of 300+ languages, as listed in Appendix A. The assignment of data to programming languages is performed based solely on file extension, similar to StarCoder (Li et al., 2023a). After language filtering, we apply four key filtering rules to filter out lower-quality code (Li et al., 2023a): (1) remove files with fewer than 25% alphabetic characters, (2) except for the XSLT language, filter out files where the string "<?xml version=" appears within the first 100 characters, (3) for HTML files, only keep files where the visible text makes up at least 20% of the HTML code and has a minimum length of 100 characters, (4) for JSON and YAML files, only keep files that have a character count ranging from 50 to 5000 characters. We also filter GitHub issues using a set of quality metrics that include removing auto-generated text, filtering out non-English issues, excluding comments from bots, and using the number of users engaged in the conversation as an indicator of quality. We also annotate each code file with license information associated with the respective repository, found via Github APIs and only keep files with permissive licenses for model training.

2.2 Exact and Fuzzy Deduplication

We adopt an aggressive deduplication strategy including both exact and fuzzy deduplication to remove documents having (near) identical code content in our training set. For exact deduplication, we first compute SHA256 hash on the document content and remove records having identical hashes. Post exact deduplication, we apply fuzzy deduplication with the goal of removing code files that may have slight variations and thereby unbiasing the data further. We apply a two-step method for this: (1) compute MinHashes of all the documents and then utilize Locally Sensitive Hashing (LSH) to group documents based on their MinHash fingerprints, (2) measure Jaccard similarity between each pair of documents in the same bucket and annotate documents except one as duplicates based on a similarity threshold of 0.7. We apply this near-deduplication process to all programming languages including GitHub issues to enhance the richness and diversity of the training dataset.

2.3 HAP, PII, Malware Filtering

To reduce the likelihood of generating hateful, abusive, or profane (HAP) language from the models, we make diligent efforts to filter HAP content from the training set. We first create a dictionary of HAP keywords and then annotate each code document with the number of occurrences of such keywords in the content including comments. We filter out documents which exceeds the HAP threshold, computed based on a distributional analysis as well as manual inspection of code files. Moreover, to protect privacy, we follow StarCoder (Li et al., 2023a) and make diligent efforts to redact Personally Identifiable Information (PII) from the training set. Specifically, we leverage the StarPII[4] model to detect IP addresses,

[2] https://huggingface.co/datasets/codeparrot/github-code-clean
[3] https://huggingface.co/datasets/bigcode/starcoderdata
[4] https://huggingface.co/bigcode/starpii

keys, email addresses, names, user names, and passwords found in the content. The PII redaction step replaces the PII text with the corresponding tokens ⟨NAME⟩, ⟨EMAIL⟩, ⟨KEY⟩, ⟨PASSWORD⟩ and change the IP address with a synthetically generated IP address, as in Li et al. (2023a). We also scan our datasets using ClamAV[5] to identify and remove instances of malware in the source code.

2.4 Natural Language Datasets

In addition to collecting code data for model training, we curate several publicly available high-quality natural language datasets for improving the model's proficiency in language understanding and mathematical reasoning. Representative datasets under this category include web documents (Stackexchange, CommonCrawl), mathematical web text (OpenWebMath; Paster et al. (2023), StackMathQA; Zhang (2024)), academic text (Arxiv, Wikipedia), and instruction tuning datasets (FLAN; Longpre et al. (2023), HelpSteer (Wang et al., 2023)). We do not deduplicate these already preprocessed natural language datasets.

3 Model Architecture

We train a series of code models of varying sizes based on the transformer decoder architecture (Vaswani et al., 2017). The model hyperparameters for these models are given in Table 1. For all model architectures, we use pre-normalization (Xiong et al., 2020): normalization applied to the input of attention and MLP blocks.

Table 1: Model configurations for Granite Code models.

Model	3B	8B	20B	34B
Batch size	2048	1024	576	532
Context length	2048	4096	8192	8192
Hidden size	2560	4096	6144	6144
FFN hidden size	10240	14336	24576	24576
Attention heads	32	32	48	48
Key-Value heads	32 (MHA)	8 (GQA)	1 (MQA)	1 (MQA)
Layers	32	36	52	88
Normalization	RMSNorm	RMSNorm	LayerNorm	LayerNorm
Activation	swiglu	swiglu	gelu	gelu
Vocab size	49152	49152	49152	49152

3B: The smallest model in the Granite-code model family is trained with RoPE embedding (Su et al., 2023) and Multi-Head Attention (Vaswani et al., 2017). This model use the swish activation function (Ramachandran et al., 2017) with GLU (Shazeer, 2020) for the MLP, also commonly referred to as swiglu. For normalization, we use RMSNorm (Zhang & Sennrich, 2019) since it's computationally more efficient than LayerNorm (Ba et al., 2016). The 3B model is trained with a context length of 2048 tokens.

8B: The 8B model has a similar architecture as the 3B model with the exception of using Grouped-Query Attention (GQA) (Ainslie et al., 2023). Using GQA offers a better tradeoff between model performance and inference efficiency at this scale. We train the 8B model with a context length of 4096 tokens.

20B: The 20B code model is trained with learned absolute position embeddings. We use Multi-Query Attention (Shazeer, 2019) during training for efficient downstream inference. For the MLP block, we use the GELU activation function (Hendrycks & Gimpel, 2023). For normalizing the activations, we use LayerNorm (Ba et al., 2016). This model is trained with a context length of 8192 tokens.

34B: To train the 34B model, we follow the approach by Kim et al. for depth upscaling of the 20B model. Specifically, we first duplicate the 20B code model with 52 layers and then

[5] https://www.clamav.net/

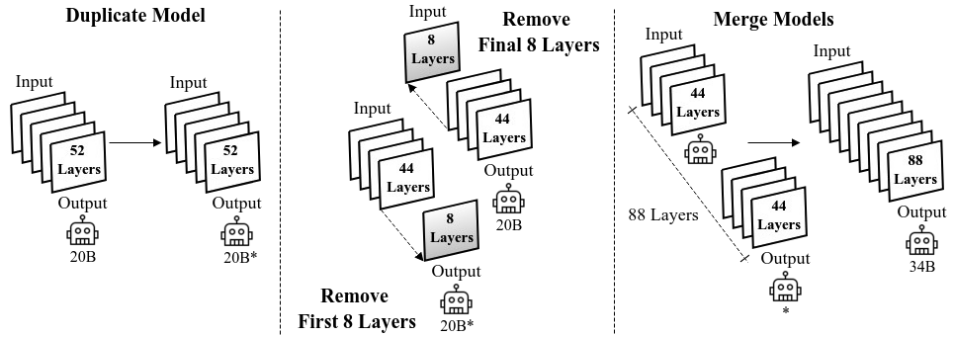

Figure 2: An overview of depth upscaling (Kim et al., 2024) for efficient training of Granite-34B-Code. We utilize the 20B model after 1.6T tokens to start training of 34B model with the same code pretraining data without any changes to the training and inference framework.

remove final 8 layers from the original model and initial 8 layers from its duplicate to form two models. Finally, we concatenate both models to form Granite-34B-Code model with 88 layers (see Figure 2 for an illustration). After the depth upscaling, we observe that the drop in performance compared to 20B model is pretty small contrary to what is observed by Kim et al.. This performance is recovered pretty quickly after we continue pretraining of the upscaled 34B model. Similar, to 20B, we use a 8192 token context during pretraining.

4 Pretraining

In this section, we provide details on two phase training (Sec. 4.1), training objectives (Sec. 4.2), optimization (Sec. 4.3) and infrastructure (Sec. 4.4) used in pretraining the models.

4.1 Two Phase Training

Granite Code models are trained on 3.5T to 4.5T tokens of code data and natural language datasets related to code. Data is tokenized via byte pair encoding (BPE, (Sennrich et al., 2015)), employing the same tokenizer as StarCoder (Li et al., 2023a). Following (Shen et al., 2024; Hu et al., 2024), we utilize high-quality data with two phases of training as follows.

- **Phase 1 (code only training)**: During phase 1, both 3B and 8B models are trained for 4 trillion tokens of code data comprising 116 languages. The 20B parameter model is trained on 3 trillion tokens of code. The 34B model is trained on 1.4T tokens after the depth upscaling which is done on the 1.6T checkpoint of 20B model.
- **Phase 2 (code + language training)**: In phase 2, we include additional high-quality publicly available data from various domains, including technical, mathematics, and web documents, to further improve the model's performance in reasoning and problem solving skills, which are essential for code generation. We train all our models for 500B tokens (80% code and 20% language data) in phase 2 training.

4.2 Training Objective

For training of all our models, we use the causal language modeling objective and Fill-In-the-Middle (FIM) (Bavarian et al., 2022) objective. The FIM objective is tasked to predict inserted tokens with the given context and subsequent text. We train our models to work with both PSM (Prefix-Suffix-Middle) and SPM (Suffix-Prefix-Middle) modes, with relevant formatting control tokens, same as StarCoder (Li et al., 2023a).

The overall loss is computed as a weighted combination of the 2 objectives:

$$L = \alpha L_{CLM} + (1 - \alpha) L_{FIM} \quad (1)$$

We emperically set $\alpha = 0.5$ during training and find that this works well in practice leading to SOTA performance on both code completion and code infilling tasks. It should be

noted that the FIM objective is only used during pretraining, however we drop it during instruction finetuning i.e we set $\alpha = 1$.

4.3 Optimization

We use AdamW optimizer (Kingma & Ba, 2017) with$\beta_1 = 0.9, \beta_2 = 0.95$ and weight decay of 0.1 for training all our Granite code models. For the phase-1 pretraining, the learning rate follows a cosine schedule starting from 3×10^{-4} which decays to 3×10^{-5} with an initial linear warmup step of 2k iterations. For phase-2 pretraining, we start from 3×10^{-4} (1.5×10^{-4} for 20B and 34B models) and adopt an exponential decay schedule to anneal it to 10% of the initial learning rate. We use a batch size of 4M-5M tokens depending on the model size during both phases of pretraining.

To accelerate training, we use FlashAttention 2 (Dao et al., 2022; Dao, 2023), the persistent layernorm kernel, Fused RMSNorm kernel (depending on the model) and the Fused Adam kernel available in NVIDIA's Apex library. We use a custom fork of NVIDIA's Megatron-LM (Shoeybi et al., 2019; Narayanan et al., 2021) for distributed training of all our models. We train with a mix of 3D parallelism: tensor parallel, pipeline parallel and data parallel. We also use sequence parallelism (Korthikanti et al., 2023) for reducing the activation memory consumption of large context length during training. We use Megatron's distributed optimizer with mixed precision training (Micikevicius et al., 2018) in BF16 (Kalamkar et al., 2019) with gradient all-reduce and gradient accumulation in FP32 for training stability.

4.4 Infrastructure

We train the Granite Code models using IBM's two supercomputing clusters, namely Vela and Blue Vela, outfitted with NVIDIA A100 and H100 GPUs, respectively. In the Vela A100 GPU cluster, each node has 2× Intel Xeon Scalable Processors with 8× 80GB A100 GPUs connected to each other by NVLink and NVSwitch. The Vela cluster adopts RoCE (RDMA over Converged Ethernet) and GDR (GPU-direct RDMA) for high-performance networking. Similarly, each node in Blue Vela cluster consists of dual 48-core Intel processors with 8× 80GB H100 GPUs. Blue Vela employs 3.2Tbps InfiniBand interconnect to facilitate seamless communication between nodes, known for their high throughput and low latency. In addition, Blue Vela employs a separate, dedicated InfiniBand Storage fabric providing 800Gbps per compute node, backed by multiple ESS6000 storage appliances. Both clusters provide a scalable and efficient infrastructure for training our models over thousands of GPUs. We estimate the carbon emissions from pretraining the Granite Code models to be ∼455 tCO_2eq, which is computed based on the total energy usage in the models and US national average carbon intensity factor of 0.423 kg CO_2eq/KWh without taking the location of data centers in consideration. The Blue Vela cluster runs on 100% renewable energy to minimize the environmental impact.

5 Instruction Tuning

Finetuning code LLMs on a variety of tasks explained via instructions has been shown to improve model usability and general performance. While there has been much progress in code instruction tuning, most of them adopt synthetically generated data from OpenAI models, which limits the model use in many enterprise applications. Thus, following OctoCoder (Muennighoff et al., 2023), we use only a combination of permissively licensed data, with an aim to enhance instruction following capabilities of our models, including logical reasoning and problem-solving skills. Specifically, Granite Code Instruct models are trained on the following types of data.

- **Code Commits Dataset**: CommitPackFT (Muennighoff et al., 2023), a filtered version of full CommitPack dataset across 92 programming languages[6];

[6]We selected 92 programming languages that are common across the original CommitPackFT and our list of 116 languages used during pretraining.

Table 2: Summary of evaluation tasks.

Task	Benchmark	Reference
Multilingual code generation	HumanEvalSynthesize	Muennighoff et al. (2023)
Multilingual code generation	MultiPL-E	Cassano et al. (2023)
Python code generation	MBPP	Austin et al. (2021)
Python code generation	MBPP+	Liu et al. (2023a)
Data science code generation	DS1000	Lai et al. (2023)
Repository-level code generation	RepoBench	Liu et al. (2023b)
Repository-level code generation	CrossCodeEval	Ding et al. (2023)
Fill-in-the-middle code completion	SantaCoder-FIM	Allal et al. (2023)
Multilingual code explanation	HumanEvalExplain	Muennighoff et al. (2023)
Multilingual code fixing	HumanEvalFix	Muennighoff et al. (2023)
Code editing	CanItEdit	Cassano et al. (2024)
Code translation	CodeLingua	Pan et al. (2024)
Code execution	CruxEval	Gu et al. (2024)
Math reasoning	MATH	Hendrycks et al. (2021)
Math reasoning	GSM8K	Cobbe et al. (2021)
Math reasoning	SAT	Azerbayev et al. (2023)
Math reasoning	OCW	Lewkowycz et al. (2022)
Function calling	BFCL	Yan et al. (2024)
Model robustness	ReCode	Wang et al. (2022)

- **Math Datasets**: MathInstruct[7] (Yue et al., 2023) and MetaMathQA (Yu et al., 2023);
- **Code Instruction Datasets**: Glaive-Code-Assistant-v3[8], Self-OSS-Instruct-SC2[9], Glaive-Function-Calling-v2[10], NL2SQL[11] and few synthetically generated API calling datasets (Basu et al., 2024);
- **Language Instruction Datasets**: High-quality datasets like HelpSteer (Wang et al., 2023), an open license-filtered version of Platypus[12] (Lee et al., 2023) including a collection of hardcoded prompts to ensure model generates correct outputs given inquiries about its name or developers.

For training, we use a cosine scheduler with 250 warmup steps, an initial learning rate 10^{-5}, and train for three epochs. Further, we add random, uniform noise with a magnitude of $\frac{\sqrt{5}}{Nh}$, where N is the sequence length and h is the embedding dimension, to the embedding vector, as proposed by Jain et al.. The additional noise improved overall answer quality of the instruction model. We use FlashAttention 2 (Dao, 2023; Dao et al., 2022) with a Padding-Free Transformer[13] implementation to reduce GPU memory usage and redundant FLOPs during finetuning. We also use full activation checkpointing (Korthikanti et al., 2023), which allows us to finetune our Granite-20B-Code models with 8K context length within a single node within a few hours on 8×A100 GPUs.

6 Evaluation

We evaluate Granite Code models on a wide variety of tasks, including code generation, code explanation, code fixing, code editing, math reasoning, etc., as shown in Table 2. We compare our models with several open-source code LLMs: StableCode (Pinnaparaju et al., 2024),

[7] We removed GSM8K-RFT and Camel-Math from MathInstruct due to unknown or NC license.
[8] https://huggingface.co/datasets/glaiveai/glaive-code-assistant-v3
[9] https://huggingface.co/datasets/bigcode/self-oss-instruct-sc2-exec-filter-50k
[10] https://huggingface.co/datasets/glaiveai/glaive-function-calling-v2
[11] https://huggingface.co/datasets/bugdaryan/sql-create-context-instruction
[12] https://huggingface.co/datasets/garage-bAInd/Open-Platypus
[13] https://huggingface.co/blog/mayank-mishra/padding-free-transformer

Code Llama (Roziere et al., 2023), StarCoder (Li et al., 2023b), StarCoder2 (Lozhkov et al., 2024), and CodeGemma[14], including recent high-performing general purpose open LLMs like Mistral (Jiang et al., 2023a) and LLama-3[15]. For all the benchmarks, we evaluate the baseline models (including ours) using the same script and environment for fair comparison.

6.1 Code Generation

6.1.1 HumanEvalSynthesize: Multilingual Code Generation in 6 Languages

While most of the prior code LLMs evaluate code generation capabilities only on Python using HumanEval (Chen et al., 2021), we adopt the challenging HumanEvalSynthesize (Muennighoff et al., 2023) benchmark in our study, which extends Python problems of Humaneval Benchmark to five additional commonly used programming languages, namely JavaScript, Java, Go, C++, Rust. We evaluate all models in a zero-shot manner using greedy decoding with completion format for the base models, and instruction template for the instruction-tuned models. In constructing prompts for instruction-tuned models, we adhere to the formats provided in their official examples. We search for a suitable prompt format in the HuggingFace model card, GitHub repository, and formal publications or technical reports.

Table 3 shows the results on of base and instruct models HumanEvalSynthesize benchmark. Granite-3B-Code-Base is the best performing small model with +3% improvement over CodeGemma-2B. Overall, among base models, Granite Code models achieve the best average performance at 7B-8B scale, 2nd best average performance in 13B-20B size models, and is very close to the best model (falls behind StarCoder2-15B by 0.1%). While CodeLlama-34B achieves better score on HumanEval Python, Granite-34B-Code-Base achieves much better performance on other languages, leading to a 4% improvement on average across 6 languages. Among the instruct models, Granite Code models consistently outperform equivalent size CodeLlama; the 3B, 8B, and 20B models even outperform CodeLlama models that are two times larger. It's worth noting that even our smaller model, Granite-3B-Code-Instruct, surpasses the performance of CodeLlama-34B-Instruct. Further, we can also see that Granite Code models outperform much larger state-of-the-art open-source general-purpose language models, including Gemma, Mixtral, and Llama 3 series models. This shows that domain-specific code models could achieve better performance and efficiency, thus making them more suitable for cost- and performance-sensitive enterprise environments.

6.1.2 MultiPL-E: Multilingual Code Generation in 18 Languages

MultiPL-E (Cassano et al., 2023) is a canonical benchmark for evaluating code models on a more diverse set of 18 different programming languages. On MultiPL-E, we compare all the base models on 18 languages, by sampling 50 completions per prompt at temperature 0.2 with top-p 0.95, as in (Lozhkov et al., 2024). Table 4 shows the results of different models on MultiPL-E. As can be seen from the table, there is no single model that works best at every language across all model sizes. In comparison to the similarly sized open-source model CodeLlama-7B, Granite-8B-Code-Base performs the best on 16/18 programming languages. Of the medium models, StarCoder2-15B performs best. Among the large models, Granite-34B-Code-Base does better than CodeLlama-34B on most languages, demonstrating its effectiveness in code generation across a diverse set of languages.

6.1.3 MBPP and MBPP+: Code Generation in Python

MBPP (Austin et al., 2021) and MBPP+ (Liu et al., 2023a) are two of the most widely studied benchmarks for evaluating code models. While the prompt for each MBPP problem includes a natural language description followed by a few tests, MBPP+ consists of 35 × more tests than the original benchmarks. We use greedy decoding and report the mean pass@1 for all the models. Table 5 summarizes the results of different base models. As we can see, Granite-3B-Code-Base significantly outperforms CodeGemma-2B but falls short of StarCoder2-3B on

[14] https://storage.googleapis.com/deepmind-media/gemma/codegemma report.pdf
[15] https://github.com/meta-llama/llama3

Table 3: Pass@1 performance on HumanEvalSynthesize benchmark (Muennighoff et al., 2023). All models are evaluated using greedy decoding with completion format for the base models, and instruction template for the instruction-tuned models.

Model	Prompt	Python	JavaScript	Java	Go	C++	Rust	Avg.
Base Models								
StarCoderBase-3B	Completion	25.6	22.6	24.4	18.3	23.2	16.5	21.8
StableCode-3B	Completion	24.4	32.3	34.1	21.3	33.5	21.3	27.8
StarCoder2-3B	Completion	27.4	36.0	**42.1**	23.8	**36.6**	**24.4**	31.7
CodeGemma-2B	Completion	**39.0**	**37.8**	37.8	13.4	33.5	20.7	30.4
Granite-3B-Code-Base	Completion	36.6	37.2	40.9	**26.2**	35.4	22.0	**33.1**
StarCoderBase-7B	Completion	26.8	28.7	31.7	22.6	28.0	22.6	26.7
CodeLlama-7B	Completion	35.4	36.0	39.0	21.3	31.1	24.4	31.2
StarCoder2-7B	Completion	38.4	43.3	48.2	**31.7**	38.4	24.4	37.4
CodeGemma-7B	Completion	41.5	48.8	54.9	26.8	**44.5**	32.3	41.5
Granite-8B-Code-Base	Completion	**43.9**	**52.4**	**56.1**	**31.7**	43.9	**32.9**	**43.5**
StarCoderBase-15B	Completion	32.3	36.6	40.2	25.6	31.1	25.6	31.9
CodeLlama-13B	Completion	41.5	42.7	51.8	26.8	40.9	23.2	37.8
StarCoder2-15B	Completion	44.5	47.0	51.8	**33.5**	**50.0**	**39.6**	**44.4**
Granite-20B-Code-Base	Completion	**48.2**	**50.0**	**59.1**	32.3	40.9	35.4	44.3
CodeLlama-34B	Completion	47.4	48.2	45.6	34.1	47.0	37.2	43.3
Granite-34B-Code-Base	Completion	**48.2**	**54.9**	**61.6**	**40.2**	**50.0**	**39.6**	**49.1**
CodeLlama-70B	Completion	55.5	55.5	65.2	40.9	55.5	43.9	52.8
Gemma-2B	Completion	20.1	23.2	19.5	13.4	18.3	8.5	17.2
Gemma-7B	Completion	33.5	41.5	43.9	26.2	35.4	26.2	34.4
Mistral-7B-v0.2	Completion	32.9	34.1	36.6	22.6	30.5	18.3	29.2
Mixtral-8x7B-v0.1	Completion	42.1	53.7	52.4	33.5	42.7	35.4	43.3
Mixtral-8x22B-v0.1	Completion	51.2	57.9	64.6	40.9	57.3	32.9	50.8
Llama-3-8B	Completion	26.2	37.8	40.2	11.0	37.2	21.3	29.0
Llama-3-70B	Completion	25.0*	51.2	62.2	21.3	53.7	37.8	41.9
Instruct Models								
CodeGemma-7B-IT	Instruction	48.4	46.0	48.4	28.6	42.2	36.0	41.6
CodeLlama-7B-Instruct	Instruction	47.0	39.0	45.7	26.8	38.4	30.5	37.9
CodeLlama-13B-Instruct	Instruction	50.6	45.1	47.0	29.9	37.8	26.2	39.4
CodeLlama-34B-Instruct	Instruction	48.8	48.8	48.8	26.2	42.7	32.3	41.3
CodeLlama-70B-Instruct	Instruction	67.8	61.6	70.7	51.2	60.4	41.5	58.9
OctoCoder-15B	Instruction	43.9	39.0	39.6	30.5	36.0	25.0	35.7
Granite-3b-Code-Instruct	Instruction	51.2	43.9	41.5	31.7	40.2	29.3	39.6
Granite-8b-Code-Instruct	Instruction	57.9	52.4	58.5	43.3	48.2	37.2	49.6
Granite-20B-Code-Instruct	Instruction	60.4	53.7	58.5	42.1	45.7	42.7	50.5
Granite-34B-Code-Instruct	Instruction	62.2	56.7	62.8	47.6	57.9	41.5	54.8
Gemma-2B-IT	Instruction	17.7	13.4	10.4	7.3	17.7	2.4	11.5
Gemma-7B-IT	Instruction	28.7	17.1	29.9	18.3	18.9	6.1	19.8
Mistral-7B-Instruct-v0.2	Instruction	39.6	32.9	36.6	22.0	33.5	19.5	30.7
Mixtral-8x7B-Instruct-v0.1	Instruction	52.4	53.0	56.1	38.4	54.9	35.4	48.4
Mixtral-8x22B-Instruct-v0.1	Instruction	70.7	69.5	75.6	55.5	69.5	48.2	64.8
Llama-3-8B-Instruct	Instruction	60.4	30.5	30.5	22.6	46.3	31.1	36.9
Llama-3-70B-Instruct	Instruction	76.2	69.5	76.2	51.8	65.2	54.3	65.5

* We could not produce reasonable results for Python generation despite many attempts using different prompts, generation parameters, precision settings.

both benchmarks. At mid parameter ranges, Granite Code models beat both CodeLlama-7B and CodeLLama-13B by a margin of ∼5% and ∼15% on average respectively. Additionally, Granite-34B-Code-Base is very competitive with CodeLlama-34B with only a difference of 0.9% on average across both benchmarks.

Table 4: Pass@1 results on MultiPL-E averaged over 50 samples for each problem. All models are evaluated at temperature 0.2 and top-p 0.95.

Model	C++	C#	D	Go	Java	Julia	JavaScript	Lua	PHP
StarCoderBase-3B	19.9	13.0	12.3	13.3	15.0	16.6	16.7	16.8	17.1
StableCode-3B	31.5	15.0	12.3	19.7	23.9	24.9	26.0	23.7	23.8
StarCoder2-3B	**32.8**	**23.4**	**24.5**	**24.2**	**27.0**	**27.5**	**29.3**	**27.6**	**28.0**
CodeGemma-2B	29.4	18.6	16.8	22.0	19.4	12.5	15.6	11.9	12.8
Granite-3B-Code-Base	31.6	21.5	22.8	22.7	25.7	27.0	27.2	26.8	27.1
StarCoderBase-7B	24.0	19.6	16.3	19.8	19.5	21.7	21.6	21.9	22.1
CodeLlama-7B	29.0	21.6	20.5	21.2	24.4	26.2	26.2	26.9	26.7
StarCoder2-7B	39.1	24.7	27.6	22.2	30.5	29.6	31.8	29.8	30.2
CodeGemma-7B	43.7	**28.2**	27.6	27.9	31.3	**34.4**	35.2	**35.3**	**35.7**
Granite-8B-Code-Base	**44.3**	21.5	**30.2**	**28.0**	**33.1**	33.7	**35.5**	33.4	33.8
StarCoderBase-15B	30.2	20.6	20.4	22.0	22.9	24.6	25.2	24.9	25.2
CodeLlama-13B	38.8	24.5	27.3	26.7	30.4	31.7	32.5	31.7	31.8
StarCoder2-15B	**47.4**	**31.7**	**35.4**	27.3	**36.6**	**37.5**	38.5	**38.5**	31.9
Granite-20B-Code-Base	43.3	30.6	16.2	**29.4**	35.9	31.8	**38.9**	30.5	**38.3**
CodeLlama-34B	45.9	31.0	**30.5**	28.4	**35.2**	**36.1**	37.0	36.4	37.1
Granite-34B-Code-Base	**46.7**	**32.8**	18.7	**33.2**	33.3	31.8	**44.5**	**38.2**	**42.5**

Model	Perl	R	Ruby	Racket	Rust	Scala	Bash	Swift	TypeScript
StarCoderBase-3B	11.0	16.2	14.2	15.3	14.4	16.4	4.2	13.2	22.4
StableCode-3B	9.7	22.5	18.7	20.7	19.1	14.6	8.4	17.5	29.5
StarCoder2-3B	13.2	**26.7**	**25.2**	**24.6**	**25.3**	21.1	**12.5**	**23.2**	**35.5**
CodeGemma-2B	2.5	11.3	9.9	10.9	11.6	25.6	1.6	10.7	1.3
Granite-3B-Code-Base	**18.8**	25.8	24.1	24.1	24.0	**26.9**	7.0	22.0	31.6
StarCoderBase-7B	16.8	21.1	19.9	20.0	20.1	21.2	7.5	18.5	27.6
CodeLlama-7B	17.4	25.7	24.7	24.2	24.9	25.5	10.0	22.8	33.7
StarCoder2-7B	17.6	29.0	27.3	27.2	27.6	22.0	13.2	25.3	37.0
CodeGemma-7B	**31.2**	**34.0**	**33.1**	**32.0**	**33.6**	**39.0**	11.0	**30.8**	**45.2**
Granite-8B-Code-Base	18.8	32.2	30.4	30.4	30.4	26.9	**13.6**	27.9	31.6
StarCoderBase-15B	16.8	23.9	22.0	22.6	22.3	28.6	11.2	20.4	32.3
CodeLlama-13B	22.5	30.1	28.8	28.4	29.0	29.7	13.8	26.6	40.6
StarCoder2-15B	**36.6**	**37.2**	**36.4**	**35.9**	**36.6**	18.7	**18.7**	**33.5**	**43.9**
Granite-20B-Code-Base	26.5	15.5	28.1	17.9	35.7	**37.5**	16.7	27.6	38.7
CodeLlama-34B	28.9	**35.4**	33.8	**33.4**	34.3	32.9	16.2	31.5	39.6
Granite-34B-Code-Base	**31.5**	25.4	**33.9**	18.2	**38.9**	**41.7**	**19.3**	**36.5**	**41.5**

6.1.4 DS1000: Data Science Tasks in Python

DS-1000 (Lai et al., 2023) is a widely studied benchmark which offers a comprehensive collection of 1,000 data science workflows across seven different libraries, from Matplotlib to TensorFlow. We use temperature 0.2 and top-p 0.95 to generate 40 samples per each library and report mean pass@1 with code completion setting for all the models up to 8B parameters. Results on DS-1000 are summarized in Table 7. Of the small models, StarCoder2-3B performs the best. Granite-3B-Code-Base is in second place, outperforming CodeGemma-2B by more than 12 points on average across 7 libraries. Granite-8B-Code-Base achieves the best average performance of 34.5% outperforming all other models of the similar parameter sizes.

The Granite Code models achieve relatively high accuracy across all sizes (e.g., outperforming CodeGemma at 2B-3B scale, StarCoder2 at 7B-8B scale and CodeLlama models with half of the sizes). This shows that our Granite Code models are not only capable of generating good code but also of using libraries more accurately in real data science workflows.

6.1.5 RepoBench, CrossCodeEval: Repository-Level Code Generation

Code generation in practice often occurs within the context of a repository rather than in isolated files. Thus, we use RepoBench (Liu et al., 2023b) and CrossCodeEval (Ding et al., 2024), to evaluate repository-level code completion capabilities of different models.

Table 5: Pass@1 on MBPP and MBPP+. Results in the table represent zero-shot evaluation using greedy decoding.

Model	MBPP	MBPP+
StarCoderBase-3B	29.4	37.8
StableCode-3B	34.8	43.3
StarCoder2-3B	**42.4**	**48.6**
CodeGemma-2B	30.4	30.8
Granite-3B-Code-Base	36.0	45.1
StarCoderBase-7B	34.8	42.1
CodeLlama-7B	39.0	42.3
StarCoder2-7B	45.4	46.7
CodeGemma-7B	**53.0**	**54.9**
Granite-8B-Code-Base	42.2	49.6
StarCoderBase-15B	37.4	46.1
CodeLlama-13B	30.6	30.1
StarCoder2-15B	**51.2**	**56.6**
Granite-20B-Code-Base	43.8	51.6
CodeLlama-34B	**48.6**	**53.6**
Granite-34B-Code-Base	47.2	53.1

Table 6: Average exact match (EM) and edit similarity (ES) on RepoBench v1.1. All models are evaluated at temperature 0.2 and top-p 0.95.

Model	Python		Java	
	EM	ES	EM	ES
StarCoderBase-3B	**29.9**	**69.3**	**36.0**	**74.1**
StableCode-3B	29.4	68.5	34.9	72.8
StarCoder2-3B	27.2	67.0	35.9	74.1
CodeGemma-2B	26.2	66.9	33.6	71.6
Granite-3B-Code-Base	27.1	66.8	34.9	73.9
StarCoderBase-7B	27.1	66.5	36.5	75.0
CodeLlama-7B	29.2	67.4	37.9	**76.6**
StarCoder2-7B	28.1	67.6	37.0	75.2
CodeGemma-7B	**36.8**	**72.7**	38.3	74.3
Granite-8B-Code-Base	31.8	69.5	**38.4**	76.4
StarCoderBase-15B	29.4	67.8	37.1	75.4
CodeLlama-13B	31.4	68.8	39.4	77.4
StarCoder2-15B	31.3	69.6	39.9	77.3
Granite-20B-Code-Base	**38.0**	**72.2**	**42.3**	**78.1**
CodeLlama-34B	34.4	70.2	40.8	**78.4**
Granite-34B-Code-Base	**35.9**	**71.5**	**42.0**	77.7

Table 7: Mean pass@1 accuracy averaged over 40 samples on DS-1000. All models are evaluated at temperature 0.2 and top-p 0.95.

Model	Matplotlib	NumPy	Pandas	PyTorch	SciPy	Scikit-Learn	TensorFlow	Avg
StarCoderBase-3B	32.1	16.8	5.3	9.2	13.2	10.5	17.2	14.2
StableCode-3B	42.5	24.5	**16.2**	15.4	13.5	20.2	27.7	22.7
StarCoder2-3B	45.5	**27.7**	**16.2**	12.9	15.8	**30.8**	22.8	**25.0**
CodeGemma-2B	30.3	17.7	5.5	4.4	10.3	2.6	4.4	10.7
Granite-3B-Code-Base	**43.3**	**27.7**	11.0	**19.1**	**21.7**	16.5	**24.4**	23.4
StarCoderBase-7B	38.0	23.0	8.2	13.1	13.7	24.5	14.6	19.1
CodeLlama-7B	46.3	21.6	13.9	12.2	17.5	16.7	20.6	21.5
StarCoder2-7B	53.6	33.3	16.9	16.2	20.6	22.2	31.9	27.8
CodeGemma-7B	**56.1**	37.2	20.9	20.5	24.5	**34.7**	31.1	32.1
Granite-8B-Code-Base	51.6	**40.0**	**23.4**	**32.4**	**22.6**	29.6	**42.2**	**34.5**

On RepoBench, we evaluate using level 2k across three settings: cross-file-first (12,000 data points), cross-file-random (5,000 data points), and in-file (7,000 data points). We report the average edit similarity and exact match across the settings. Following Liu et al. (2023b), we set generation temperature to 0.2 and the top-p sampling parameter to 0.95 for all models. We constrain the models to generate a maximum of 64 new tokens per prompt, and the first non-empty and non-comment line of the output was selected as the prediction.

For CrossCodeEval, following Ding et al. (2024), we use a max sequence length of 2k using the retrieve-and-generate (RG) method with OpenAI's ada embedding. We set the maximum cross-file context to 512 tokens and the max generation token to 50 tokens for all the models. We use the uniform prompt formatting in the original implementation, with a temperature of 0.2 and top-p of 0.95 for all model generations. Max sequence length was set to 8,192 for all models, with the exception of Granite-3B-Code-Base (2,048) and Granite-8B-Code-Base (4,096), given their respective context lengths.

Table 6 shows the performance of different models on RepoBench v1.1. Granite-3B-Code-Base demonstrates notable performance among the smaller models, with StarCoderBase-3B achieving the leading performance metrics. Among the medium models, Granite-8B-Code-Base shows very strong performance on Java, while ranks second best one in Python, with CodeGemma-7B being the best performing on both metrics. Among larger models, Granite-20B-Code not only outperforms StarCoder2-15B but also CodeLlama-34B on all 4 metrics across both programming languages. This demonstrates strong repository-level

Table 8: CrossCodeEval (Ding et al., 2023) evaluation results. We report Code Match (Edit Similarity) and Identifier Match (F1) results for four languages.

Model	Python		Java		TypeScript		C#	
	Code ES	ID F1	Code ES	ID F1	Code ES	ID F1	Code ES	ID F1
StarCoderBase-3B	63.5	53.8	63.3	55.7	44.2	40.8	65.3	45.0
StableCode-3B	65.3	56.1	**68.2**	**61.0**	**60.9**	**55.7**	59.9	41.7
StarCoder2-3B	**65.5**	**56.7**	64.8	57.3	44.7	41.3	**66.0**	**47.5**
CodeGemma-2B	60.5	50.6	55.1	46.4	55.6	49.0	44.2	27.9
Granite-3B-Code-Base	65.1	56.0	64.1	56.6	43.2	39.4	65.9	46.6
StarCoderBase-7B	65.3	56.3	65.4	58.0	46.2	43.2	66.1	47.2
CodeLlama-7B	64.9	55.4	65.0	57.8	62.1	56.9	65.1	46.9
StarCoder2-7B	66.5	57.5	**67.0**	**59.8**	46.9	43.3	**67.2**	48.6
CodeGemma-7B	**68.1**	**59.3**	65.9	59.6	**63.5**	**58.5**	56.2	42.1
Granite-8B-Code-Base	66.3	57.8	66.5	59.0	45.1	42.0	66.6	**48.7**
StarCoderBase-15B	66.0	57.3	67.0	60.2	46.6	43.3	66.4	47.7
CodeLlama-13B	66.2	57.4	66.8	59.5	63.5	**58.6**	65.6	48.3
StarCoder2-15B	68.1	**59.7**	68.1	**61.5**	47.0	43.7	**68.5**	**51.0**
Granite-20B-Code-Base	**68.2**	58.1	**68.4**	60.4	**48.3**	43.1	67.5	48.4
CodeLlama-34B	**69.3**	**59.6**	68.2	**61.1**	**64.4**	**56.9**	67.2	**49.6**
Granite-34B-Code-Base	68.3	58.5	**68.6**	60.8	48.9	43.6	**67.5**	49.1

Table 9: Exact-match on FIM-task (Allal et al., 2023). All models are evaluated using greedy decoding with maximum new tokens set to 512.

Model	Java	JavaScript	Python	Avg.
StarCoderBase-3B	76.0	68.5	53.6	66.0
StableCode-3B	64.2	**74.5**	59.6	66.1
StarCoder2-3B	76.0	73.5	59.4	69.6
Granite-3B-Code-Base	**79.7**	71.6	**61.8**	**71.0**
StarCoderBase-7B	81.1	74.5	62.0	72.5
StarCoder2-7B	82.1	78.4	61.5	74.0
Granite-8B-Code-Base	**83.6**	**79.9**	**66.3**	**76.6**
StarCoderBase-15B	74.6	74.6	63.1	70.8
StarCoder2-15B	61.1	54.8	48.4	54.8
Granite-20B-Code-Base	79.4	**82.2**	**66.8**	**76.1**
Granite-34B-Code-Base	**80.8**	79.4	67.9	76.0

code generation capabilities of the Granite Code models despite being not trained with repo-level file packing as in (Lozhkov et al., 2024; CodeGemma Team et al., 2024); we leave this as an interesting future work to further improve performance of our models.

Results on CrossCodeEval are shown in Table 8. As can be seen from the table, among the similar sized models, CodeGemma-7B is best on Python and TypeScript, while StarCoder2-7B performs best on Java and C#. Likewise, Granite-20B-Code-Base outperforms CodeLlama-13B on 3 programming languages (Python, Java, C#), while falls behind on TypeScript. Across all model sizes and programming languages, there is no single model that is best at all the metrics, similar to the findings in MultiPL-E. This indicates that achieving uniformly high performance across all programming languages remains challenging.

6.1.6 FIM: Infilling Evaluations

Granite Code models are trained for code completion purposes using FIM objective, as described in Sec. 4.2. We use SantaCoder-FIM benchmark (Allal et al., 2023), for infilling evaluations which tests the ability of models to fill in a single line of code in Python, JavaScript, and Java solutions to HumanEval. We use greedy decoding and report the mean exact match for all the models. Table 9 shows that Granite Code models significantly outperforms StarCoder and StarCoder2 across all model sizes, demonstrating it to be

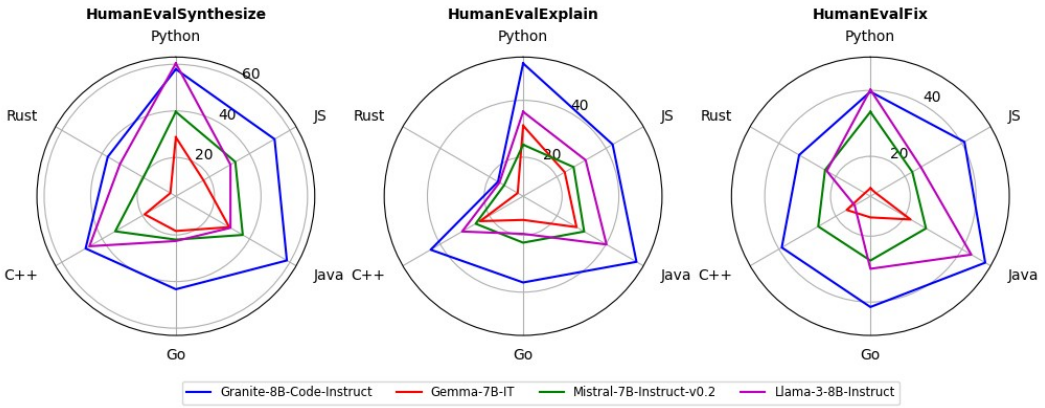

Figure 3: Performance of Granite-8B-Code-Instruct, Mistral-7B-Instruct-v0.2, Gemma-7B-IT, and Llama-3-8B-Instruct on HumanEvalPack. Best viewed in color.

excellent well-rounded models for code completion use cases. Moreover, we observe no performance improvement in scaling the model sizes from 8B to 34B, indicating that smaller models are often more suitable for FIM code completion tasks.

6.2 Code Explanation and Fixing

While most of the prior code LLMs primarily focus on evaluating performance using code generation benchmarks, users may want to use these models in other challenging scenarios beyond synthesis like explaining and fixing codes. Thus, following (Muennighoff et al., 2023), we test the performance of different code models on the code explanation and fixing variants of HumanEvalPack benchmark, spanning 6 different programming languages. For both HumanEvalExplain and HumanEvalFix, we evaluate all models in a zero-shot manner using greedy decoding with completion format for the base models, and instruction template for the instruction-tuned models.

The results of the HumanEvalExplain benchmark are shown in Table 10. Granite Code Base models significantly outperform other SOTA base code LLMs, including StarCoder2 and CodeGemma, by a large margin. Interestingly, Granite-8B-Code-Base beats CodeLlama-34B by 9.3% on average, while being close to CodeLlama-70B. We attribute this performance to our data mixture and base model training decisions. After instruction tuning, performance of all the base models significantly improves across languages. Among code instruct models, Granite-34B-Code-Instruct performs the best reaching the average score of 41.9%, which is very close of 41.1% score of CodeLlama-70B-Instruct. Remarkably, CodeGemma-7B-IT gains the most improvement from instruction tuning but still falls behind Granite-8b-Code-Instruct by 2.5% on average. Mixtral-8x22B-Instruct-v0.1 performs best among all models benchmarked with a significant margin, indicating the potential advantage of bigger models and training on general natural language data could potentially help on this task.

Table 11 reports the results on HumanEvalFix. Like HumanEvalExplain, Granite Code models base models significantly outperform other base models. Notably, Granite-8B-Code-Base again shows impressive performance, making it close to CodeLlama-70B and Llama-3-70B. After instruction tuning, we observe a performance improvement on almost all models. Notably, our 8B and 20B instruct models achieve the best performance among models with less that has less than 34B parameters. However, we see a significant performance improvement (about 10 points) after moving to larger models with more than 34B parameters. Among large instruct models, Granite-34B-Code-Instruct performs similarly to other models with at least twice the parameters, thus having a better cost and performance balance.

Figure 3 compares the performance of Granite-8B-Code-Instruct with state-of-the-art open-source instruction-tuned general LLMs. Granite-8B-Code-Instruct consistently outperforms the compared models, emphasizing the need for domain-specific code models. To summa-

Table 10: Pass@1 performance on HumanEvalExplain.

Model	Prompt	Python	JavaScript	Java	Go	C++	Rust	Avg.
Base Models								
StarCoderBase-3B	Completion	11.0	10.4	14.6	11.0	13.4	11.0	11.9
StableCode-3B	Completion	11.0	7.9	22.0	4.3	14.6	14.0	12.3
StarCoder2-3B	Completion	12.2	13.4	19.5	6.7	14.0	12.8	13.1
CodeGemma-2B	Completion	20.7	15.9	20.7	12.8	17.7	15.9	17.3
Granite-3B-Code-Base	Completion	**25.0**	**18.9**	**29.9**	**17.1**	**26.8**	**14.0**	**21.9**
StarCoderBase-7B	Completion	14.0	17.1	17.7	10.4	17.1	12.8	14.8
CodeLlama-7B	Completion	11.0	14.0	16.5	9.8	17.7	14.6	13.9
StarCoder2-7B	Completion	4.9	12.8	22.0	4.9	22.0	14.6	13.5
CodeGemma-7B	Completion	13.1	13.8	2.0	8.0	18.6	18.9	12.4
Granite-8B-Code-Base	Completion	**23.5**	**32.3**	**25.0**	**23.2**	**28.0**	**19.5**	**26.4**
StarCoderBase-15B	Completion	9.8	15.2	**24.4**	9.1	20.1	13.4	15.3
CodeLlama-13B	Completion	13.4	14.0	23.2	9.8	15.9	13.4	15.0
StarCoder2-15B	Completion	**20.1**	**19.5**	7.3	9.8	23.8	**21.3**	17.0
Granite-20B-Code-Base	Completion	17.1	18.3	23.2	**10.4**	**25.6**	18.3	**18.8**
CodeLlama-34B	Completion	11.6	18.3	22.0	9.8	20.1	20.7	17.1
Granite-34B-Code-Base	Completion	**42.7**	**26.2**	**47.0**	**26.8**	**36.6**	**25.0**	**34.1**
CodeLlama-70B	Completion	24.4	30.5	43.9	19.5	31.1	20.1	28.2
Gemma-2B	Completion	9.8	9.8	14.6	7.9	14.0	9.1	10.9
Gemma-7B	Completion	10.4	18.3	19.5	9.8	18.3	14.0	15.0
Mistral-7B-v0.2	Completion	22.0	23.8	34.8	16.5	14.6	12.8	20.7
Mixtral-8x7B-v0.1	Completion	17.1	18.3	35.4	19.5	18.9	15.2	20.7
Mixtral-8x22B-v0.1	Completion	29.9	20.1	40.2	17.7	22.0	17.7	24.6
Llama-3-8B	Completion	15.2	14.0	18.9	5.5	18.3	8.5	13.4
Llama-3-70B	Completion	12.2	18.9	20.7	9.1	16.5	15.9	15.6
Instruct Models								
CodeGemma-7B-IT	Instruction	48.2	40.9	51.8	31.1	33.5	25.0	38.4
CodeLlama-7B-Instruct	Instruction	29.9	29.9	32.9	19.5	25.0	13.4	25.1
CodeLlama-13B-Instruct	Instruction	38.4	28.0	36.0	22.6	26.8	14.6	27.7
CodeLlama-34B-Instruct	Instruction	42.1	31.1	39.6	20.1	31.7	17.1	30.3
CodeLlama-70B-Instruct	Instruction	47.0	40.2	54.9	34.1	46.3	23.8	41.1
OctoCoder-15B	Instruction	37.8	26.8	31.1	18.3	22.6	14.0	25.1
Granite-3b-Code-Instruct	Instruction	39.6	26.8	39.0	14.0	23.8	12.8	26.0
Granite-8b-Code-Instruct	Instruction	53.0	42.7	52.4	36.6	43.9	16.5	40.9
Granite-20B-Code-Instruct	Instruction	44.5	42.7	49.4	32.3	42.1	18.3	38.2
Granite-34B-Code-Instruct	Instruction	53.0	45.1	50.6	36.0	42.7	23.8	41.9
Gemma-2B-IT	Instruction	9.8	12.8	13.4	8.5	13.4	3.0	10.2
Gemma-7B-IT	Instruction	31.1	23.2	28.0	14.6	23.8	8.5	21.5
Mistral-7B-Instruct-v0.2	Instruction	24.4	26.8	31.1	22.6	25.6	14.0	24.1
Mixtral-8x7B-Instruct-v0.1	Instruction	47.0	40.9	48.2	28.0	32.9	25.0	37.0
Mixtral-8x22B-Instruct-v0.1	Instruction	67.1	56.7	67.7	44.5	64.0	39.6	56.6
Llama-3-8B-Instruct	Instruction	36.0	31.7	40.2	19.5	31.1	15.9	29.1
Llama-3-70B-Instruct	Instruction	50.6	47.6	57.9	34.8	48.2	33.5	45.4

rize, these results show that both our base and instruct models are capable of generating good code but also in code fixing and explanation, demonstrating their ability to solve diverse coding tasks in enterprise software development.

6.3 Code Editing and Translation

CanItEdit is a recent benchmark designed to evaluate code LLMs on instructional code editing tasks. The benchmark contains 105 hand-crafted Python programs where each problem consists of a code snippet accompanied by an instruction of two types: descriptive or lazy. The goal is to modify the code according to the instruction; both lazy and descriptive instructions should lead to the same edit. Following Cassano et al. (2024), we compare different instruction-tuned models using their corresponding instruction format, by random sampling with a temperature of 0.2 and a top-p of 0.95, with 20 completions per problem.

Table 11: Pass@1 performance on HumanEvalFix.

Model	Prompt	Python	JavaScript	Java	Go	C++	Rust	Avg.
Base Models								
StarCoderBase-3B	Completion	12.2	9.8	6.1	7.9	1.8	0.6	6.4
StableCode-3B	Completion	11.0	7.3	20.1	10.4	12.8	1.2	10.5
StarCoder2-3B	Completion	**18.3**	15.9	12.8	12.8	5.5	0.6	11.0
CodeGemma-2B	Completion	4.3	7.3	3.0	9.8	1.8	0.0	4.4
Granite-3B-Code-Base	Completion	**18.3**	**23.2**	**29.9**	**24.4**	**16.5**	**3.7**	**19.3**
StarCoderBase-7B	Completion	15.9	21.3	17.1	14.6	5.5	0.6	12.5
CodeLlama-7B	Completion	15.9	14.0	23.8	15.2	5.5	11.0	14.2
StarCoder2-7B	Completion	5.5	13.4	15.9	11.0	7.3	0.6	8.9
CodeGemma-7B	Completion	8.5	5.5	20.1	14.6	7.9	3.7	10.1
Granite-8B-Code-Base	Completion	**22.6**	**35.4**	**38.4**	**37.2**	**28.7**	**15.2**	**29.6**
StarCoderBase-15B	Completion	10.4	17.7	17.1	18.9	9.8	3.7	12.9
CodeLlama-13B	Completion	6.1	9.1	17.1	9.8	6.1	10.4	9.5
StarCoder2-15B	Completion	9.1	18.9	25.0	37.2	25.0	16.5	21.9
Granite-20B-Code-Base	Completion	23.2	23.8	14.6	26.2	15.2	3.0	17.7
CodeLlama-34B	Completion	14.0	20.7	20.1	26.2	31.1	6.1	19.7
Granite-34B-Code-Base	Completion	**20.1**	**30.5**	**40.9**	**34.1**	**39.0**	**12.2**	**29.5**
CodeLlama-70B	Completion	12.8	31.1	41.5	42.1	38.4	31.1	32.8
Gemma-2B	Completion	1.2	2.4	4.3	2.4	1.2	0.0	1.9
Gemma-7B	Completion	1.8	1.2	26.2	7.3	6.7	1.2	7.4
Mistral-7B-v0.2	Completion	3.0	2.4	6.1	9.1	5.5	0.6	4.5
Mixtral-8x7B-v0.1	Completion	11.0	12.8	18.3	25.0	15.9	3.0	14.3
Mixtral-8x22B-v0.1	Completion	17.1	30.5	23.8	30.5	32.3	11.0	24.2
Llama-3-8B	Completion	2.6	3.7	5.5	3.7	1.3	0.9	2.9
Llama-3-70B	Completion	31.7	33.5	39.0	28.0	28.7	12.8	28.9
Instruct Models								
CodeGemma-7B-IT	Instruction	46.3	45.7	52.4	48.2	43.9	38.4	45.8
CodeLlama-7B-Instruct	Instruction	19.5	18.9	13.4	14.6	6.1	4.3	12.8
CodeLlama-13B-Instruct	Instruction	18.9	18.9	24.4	22.6	9.8	0.0	15.8
CodeLlama-34B-Instruct	Instruction	37.8	28.0	37.2	24.4	24.4	17.7	28.2
CodeLlama-70B-Instruct	Instruction	64.6	52.4	57.3	51.8	51.2	40.9	53.0
OctoCoder-15B	Instruction	28.0	28.7	34.1	26.8	25.0	15.9	26.4
Granite-3b-Code-Instruct	Instruction	26.8	28.0	33.5	27.4	31.7	16.5	27.3
Granite-8b-Code-Instruct	Instruction	39.6	40.9	48.2	41.5	39.0	32.9	40.4
Granite-20B-Code-Instruct	Instruction	43.9	43.9	45.7	41.5	41.5	29.9	41.1
Granite-34B-Code-Instruct	Instruction	54.9	47.6	55.5	51.2	47.0	45.1	50.2
Gemma-2B-IT	Instruction	18.9	15.9	25.6	13.4	15.9	10.4	16.7
Gemma-7B-IT	Instruction	10.3	9.8	22.0	14.3	16.2	9.8	13.7
Mistral-7B-Instruct-v0.2	Instruction	33.5	22.6	27.4	27.4	26.2	23.8	26.8
Mixtral-8x7B-Instruct-v0.1	Instruction	44.5	37.8	47.6	38.4	39.0	28.7	39.3
Mixtral-8x22B-Instruct-v0.1	Instruction	59.1	53.7	66.5	55.5	56.1	45.7	56.1
Llama-3-8B-Instruct	Instruction	40.2	25.6	43.3	29.9	13.4	23.2	29.3
Llama-3-70B-Instruct	Instruction	57.3	51.2	54.3	51.8	50.6	49.4	52.4

Table 12: Pass@1 and ExcessCode performance on CanItEdit. Pass@1 assesses functional correctness, and ExcessCode assesses conciseness and precision of code edits.

Model	Descriptive		Lazy	
	Pass@1 (⇑)	ExcessCode (⇓)	Pass@1 (⇑)	ExcessCode (⇓)
CodeGemma-7B-IT	31.57	1.03	25.28	0.58
CodeLlama-7B-Instruct	33.23	0.80	24.80	0.45
Granite-8B-Code-Instruct	**39.72**	**0.30**	**32.38**	**0.08**
CodeLlama-13B-Instruct	**41.09**	**0.20**	**29.85**	0.70
Granite-20B-Code-Instruct	40.52	0.30	29.72	**0.02**
CodeLlama-34B-Instruct	46.28	**0.05**	**37.90**	0.43
Granite-34B-Code-Instruct	**50.28**	0.25	37.28	**0.05**

Table 13: Performance of different instruction-tuned models on CodeLingua (Pan et al., 2024). We report Pass@1 for translation across five languages.

Source Language	C				C++				Go			
Target Language	C++	Go	Java	Py	C	Go	Java	Py	C	C++	Java	Py
Granite-3B-Code-Instruct	78.5	43.5	64.5	38.5	43.5	25.5	68	46.5	46.5	62.5	2.5	20.0
CodeGemma-7B-IT	**79.5**	**48.0**	60.5	44.0	29.0	**32.0**	58.0	39.0	31.0	49.5	55.5	33.5
CodeLlama-7B-Instruct	18.0	3.0	59.5	31.0	9.0	13.0	54.5	28.0	5.5	31.5	0.0	5.5
Granite-8B-Code-Instruct	78.5	18.5	**72.0**	**57.0**	**54.0**	24.5	**74.0**	**56.5**	**50.0**	**65.5**	**58.0**	**52.5**
CodeLlama-13B-Instruct	2.5	41.5	60.0	2.5	0.0	23.0	59.0	11.5	1.5	2.5	21.0	27.5
OctoCoder-15B	1.5	39.0	1.0	1.0	0.0	29.0	4.0	0.5	0.5	6.5	0.0	37.5
Granite-20B-Code-Instruct	**81.5**	**50.0**	**71.5**	**39.0**	**55.5**	**31.5**	**70.5**	**40.5**	**55.0**	**63.0**	**58.0**	**55.5**
CodeLlama-34B-Instruct	2.5	**49.0**	60.0	28.5	2.0	**26.0**	60.5	33.0	0.5	3.0	38.5	18.0
Granite-34B-Code-Instruct	**83.0**	15.5	**74.5**	**54.5**	**61.0**	25.0	**74.0**	**42.5**	**58.0**	**62.0**	**68.0**	**66.0**

Source Language	Java				Python			
Target Language	C	C++	Go	Py	C	C++	Go	Java
Granite-3B-Code-IT	34.2	63.2	20.1	44.1	18.8	45.6	1.2	15.0
CodeGemma-7B-IT	33.7	31.3	10.3	31.3	22.9	33.4	4.6	**44.5**
CodeLlama-7B-Instruct	5.3	17.8	15.8	11.6	2.55	18.0	**14.7**	28.2
Granite-8B-Code-Instruct	**49.6**	**57.3**	**17.1**	**57.9**	**41.3**	**55.2**	10.5	39.7
CodeLlama-13B-Instruct	0.0	1.25	7.6	34.2	0.0	0.9	9.6	13.1
OctoCoder-15B	0.3	3.2	18.8	22.9	0.0	2.65	16.8	32.2
Granite-20B-Code-Instruct	**44.8**	**52.8**	**33.4**	**34.3**	**41.8**	**48.4**	**26.1**	**53.4**
CodeLlama-34B-Instruct	0.0	1.8	9.8	20.6	0.5	1.0	14.1	10.0
Granite-34B-Code-Instruct	**55.0**	**65.9**	**28.5**	**56.3**	**45.7**	**53.7**	**34.1**	**61.3**

From Table 12, we make the following observations on the performance of different models on CanItEdit benchmark. It shows that Granite Code models have better pass rates, as well as less presence of unnecessary code changes, compared to CodeGemma and CodeLlama. This result shows that Granite Code models can better understand users' intentions and make accurate changes to the existing code in practical situations.

CodeLingua (Pan et al., 2024) is a dataset designed to test model's capabilities in code translation. It contains two sets of programs: one containing 251 programs in Java and 251 prgrams in Python sampled from Avatar (Ahmad et al., 2021), and one from CodeNet Puri et al. (2021) containing 250 programs for each of five languages: C, C++, Go, Java and Python. For each program a set of unit tests in the form of input and expected outputs is provided. The task consists in translating each program from the source language to five target languages (the ones sampled from CodeNet). Pass@1 is used as the metric to evaluate translation accuracy. For every generation, we used greedy decoding and the suggested prompt format for each instruction tuned model. For base models, or cases where the instruction format was not specified, we used the default prompt from the dataset. Basic post-processing is applied to each generation, to remove generation artifacts such as repetition of the input instruction, source language code, target language name and formatting tokens (``` , for example).

Table 13 shows the results on the CodeLingua benchmark. For the source languages C, C++ and Go the results reported in the table are taken directly from the runs on Codenet, whereas for Java and Python the results are reported as the average of the runs on Avatar and CodeNet. We report the numbers of Octocoder and CodeLlama from the CodeLingua leaderboard [16]. The Granite Code models performs comparably to CodeGemma. It is worth noticing that the correctness of the translation is not only due to the code generated by the model, but also by the extra metadata and explanation provided as part of the answer. We tested instruction tuned models, as we observed that base models often struggle to understand the request itself to translate code. Instruct models, on the other hand, tend to add additional information besides the translated code as part of the generations. The CodeLLama family seems to suffer especially from this issue, as post-processing the

[16] https://codetlingua.github.io/leaderboard.html

generations to extract only the relevant code constitutes a non-trivial task. The CodeGemma and Granite Models on the other hand, produce a nicely formatted output that can be easily parsed. Interestingly, Go seems to be the hardest target language to translate to, while C is the source language with the highest translation success rate from, for the Granite models.

6.4 Code Reasoning, Understanding and Execution

Table 14: Performance on the CRUXEval benchmark. We use temperature 0.2 for pass@1 and temperature 0.8 for pass@5, both using 10 samples, as in (Gu et al., 2024).

Model	CRUXEval-I		CRUXEval-O	
	Pass1	Pass5	Pass1	Pass5
StarCoderBase-3B	27.5	44.9	27.0	41.8
StableCode-3B	**33.5**	**54.2**	**32.1**	**44.0**
StarCoder2-3B	32.1	50.3	34.0	48.4
CodeGemma-2B	29.6	45.5	32.1	45.5
Granite-3B-Code-Base	30.6	50.9	31.4	35.2
StarCoderBase-7B	29.8	47.5	32.1	44.2
CodeLlama-7B	36.2	53.7	34.0	48.5
StarCoder2-7B	34.2	53.8	35.8	49.7
CodeGemma-7B	**42.6**	**60.9**	**43.9**	**56.7**
Granite-8B-Code-Base	36.0	55.8	36.1	50.3
StarCoderBase-15B	31.0	49.2	34.4	47.4
CodeLlama-13B	42.2	61.8	39.9	55.1
StarCoder2-15B	**47.4**	**68.3**	**46.7**	**59.2**
Granite-20B-Code-Base	39.1	59.0	37.5	51.7
CodeLlama-34B	**47.8**	**65.6**	42.5	**56.5**
Granite-34B-Code-Base	43.3	61.3	**44.8**	55.1

CRUXEval (Gu et al., 2024) is a benchmark of 800 Python functions and input-output pairs, consisting of two tasks: CRUXEval-I (input prediction) and CRUXEval-O (output prediction). We use temperature 0.2 to report pass@1 and temperature 0.8 to report pass@5, both using 10 samples, as in Lozhkov et al. (2024); Gu et al. (2024). Table 14 shows that Granite Code models perform competitively with other models. Granite-3B-Code-Base outperforms CodeGemma-2B on CRUXEval-I but lags behind on CRUXEval-O. Interestingly, there is not a single model which performs consistently best at 3B parameters. However, at 7B-8B parameters, CodeGemma-7B outperforms all the models on both tasks. For the large models, Granite-34B-Code-Base lags behind CodeLlama-34B on CRUXEval-I but outperforms on CRUXEval-O. Performance on both CRUXEval-I and CRUXEval-O increases as we scale the size of the Granite Code models from 3B to 34B parameters, demonstrating the advantage of larger models for code reasoning and execution tasks.

6.5 Math Reasoning

We use the following four widely used benchmarks to assess the mathematical reasoning capabilities of Granite-8B-Code-Base and various 7B-8B baseline models:

- **MATH** (Hendrycks et al., 2021): a dataset from high-school math competitions; we use the 4-shot experiment setting from Gao et al. (2023);
- **GSM8K** (Cobbe et al., 2021): a dataset of middle-school level math word problems; we use the 5-shot experiment setting from Gao et al. (2023);
- **SAT** (Azerbayev et al., 2023): a dataset consisting of the 32 math questions with no figures from the May 2023 College Board SAT examination; we use the same experiment setting from Azerbayev et al. (2023)
- **OCW** (Lewkowycz et al., 2022): a collection of undergraduate-level STEM problems harvested from MIT's OpenCourseWare; we use the 4-shot experiment setting from Azerbayev et al. (2023).

Table 15: Performance on 4 chain-of-thought math tasks and 2 tool-aided math tasks.

Model	MATH	GSM8K	SAT	OCW	MATH+Py	GSM8K+Py
StarCoderBase-7B	2.4	3.8	18.7	2.2	18.2	15.6
CodeLlama-7B	4.1	11.9	12.5	2.9	20.8	26.8
StarCoder2-7B	10.4	27.2	37.5	4.8	28.7	39.4
CodeGemma-7B	21.8	49.0	53.1	6.9	31.1	60.9
Granite-8B-Code-Base	21.4	**61.9**	62.5	8.8	**35.4**	**63.1**
Gemma-7B	**24.1**	53.3	**75.0**	7.3	27.4	52.9
Mistral-7B-v0.2	12.8	37.2	53.1	5.8	25.7	45.6
Llama-3-8B	15.6	49.8	34.4	**9.9**	0.0*	2.4*
Llemma-7B	17.3	33.7	59.4	7.0	25.6	40.8

* We noticed that Llama-3-8B-Base tends to generate invalid programs given the same prompts as the other model, resulting in very low scores on MATH+Py and GSM8K+Py tasks. Similar issues have been observed in our Python code generation experiment.

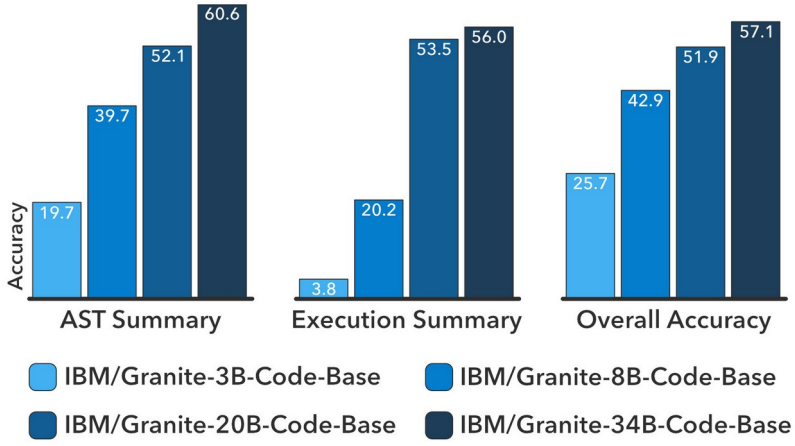

Figure 4: Performance of Granite Code models on Berkeley Function-Calling Leaderboard. Overall accuracy keeps increasing with increase in model sizes, indicating the advantage of large models for function calling abilities. Best viewed in color.

Following Azerbayev et al. (2023), we also evaluate models on two tasks that involve solving problems with access to computational tools:

- **MATH+Py** solving MATH task by writing a Python program that uses built-in numeric operations, the `math`module, and `SymPy`; we use the 5-shot prompt and experiment setting from Azerbayev et al. (2023);
- **GSM8K+Py** solving GSM8K task by writing a Python program that executes to generate an integer answer; we use the 8-shot prompt and experiment setting from Azerbayev et al. (2023).

Table 15 summarizes the results. Despite not being specifically tuned for mathematical reasoning, Granite-8B-Code-Base shows impressive reasoning ability, outperforming most existing 7B to 8B models. While other models may be particularly strong on a few tasks, our model consistently achieves top-1 or top-2 performance on all tasks.

6.6 Calling Functions and Tools

We adopt Berkeley Function-Calling Leaderboard (BFCL) (Yan et al., 2024), to evaluate LLM's ability to call functions and tools. BFCL is a function-calling dataset with 1700 functions across 4 categories: simple, multiple, parallel, and parallel multiple function calls -

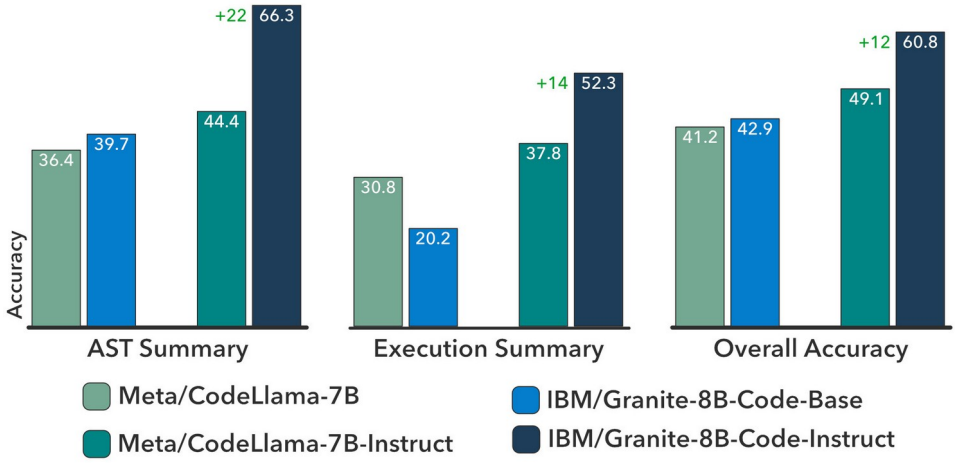

Figure 5: Granite-8B-Code vs CodeLlama-7B on Berkley Function-Calling Leaderboard. Granite-8B-Code (Base/Instruct) consistently outperforms CodeLlama-7B (Base/Instruct) on all three metrics. Best viewed in color.

each differing in the number of potential functions the model has access to and the number of output functions the model has to generate. We use two popular methods to evaluate the accuracy of the model-generated answers: AST evaluation based on Abstract Syntax Tree (AST) based metric for fuzzy evaluation of output, and Executable evaluation to match the outputs of model-generated and ground-truth functions.

Figure 4 shows the results of different Granite Code models on BFCL benchmark. As can be seen from the figure, overall accuracy improves from 25.65% to 57.12% for Granite-3B-Code-Base to Granite-34B-Code-Base, showing the effectiveness of model scaling in function (tool) calling capabilities. We also compare Granite-8B-Code with CodeLlama-7B in Figure 5 and find that Granite-8B-Code-Instruct beats CodeLlama-7B-Instruct by 22%, 14% and 12% on AST Summary, Execution Summary and Overall accuracy respectively. Additionally, Figure 5 shows that instruction tuning consistently improves performance of both base models, with more noticeable improvements in Granite Code models. E.g., +17.88% in overall accuracy from Granite-8B-Code-Base to Granite-8B-Code-Instruct, indicating the effectiveness of our well-curated data mixture in finetuning base models.

6.7 Model Robustness

While the performance on canonical code generative tasks is essential, we argue that the evaluation of practical robustness is also necessary to characterize different models systematically. We therefore consider benchmarking the robustness of code synthesis, one of the most representative downstream tasks of source code. ReCode (Wang et al., 2022) provides 30 different general perturbations on docstrings, function names, and codes to evaluate the robustness of code-generation models. We use the perturbed version of the HumanEval benchmark using greedy generation with 5 seeds, as recommended in (Wang et al., 2022).

Table 16 shows the worst-case RP@1 of different models for each perturbation category. While Granite-3B-Code-Base consistently outperforms CodeGemma-2B, Granite-8B-Code-Base lags behind CodeGemma-7B on all categories. Granite Code models obtains much better performance compared to CodeLlama models, showing its generalization in a robust way at every sizes. Our largest model, Granite-34B-Code-Base consistently outperforms CodeLlama-34B on all four categories. This indicates that Granite-34B-Code-Base has more capacity to deal with unseen instances and perturbations. In general, we also observe higher RP@1 for larger models within the Granite Code family (e.g., improved from 40.1% to 52.0% for Granite-3B-Code-Base to Granite-34B-Code-Base on average across all perturbations), showing that larger model helps improve worst-case robustness.

Table 16: RP@1 performance on the Recode benchmark. Following (Wang et al., 2022), we use the perturbed version of the HumanEval benchmark with greedy sampling for all the models to eliminate randomness effect and enable fair comparison.

Model	Docstring	Function	Syntax	Format
StarCoderBase-3B	12.3	11.4	17.2	24.2
StableCode-3B	22.8	25.8	37.1	46.4
StarCoder2-3B	**28.6**	29.7	**49.6**	**57.6**
CodeGemma-2B	12.3	11.4	17.2	24.2
Granite-3B-Code-Base	28.2	**30.0**	45.8	56.3
StarCoderBase-7B	23.7	25.3	38.2	47.1
CodeLlama-7B	24.7	27.6	43.0	53.1
StarCoder2-7B	27.6	30.4	45.8	57.5
CodeGemma-7B	**32.3**	**37.8**	**55.3**	**64.3**
Granite-8B-Code-Base	25.5	30.9	49.9	60.5
StarCoderBase-15B	26.6	30.7	44.3	52.2
CodeLlama-13B	25.8	29.7	50.6	60.3
StarCoder2-15B	**36.9**	**43.9**	**60.4**	**70.2**
Granite-20B-Code-Base	35.2	43.0	55.1	63.5
CodeLlama-34B	33.1	38.0	54.7	64.4
Granite-34B-Code-Base	**36.3**	**44.4**	**59.2**	**68.2**

7 Conclusion

We presented a family of decoder-only Granite Code models ranging in size from 3 to 34 billion parameters that are highly versatile in their ability to accomplish a wide range of tasks from code generation to fixing bugs, explaining and documenting code, maintaining repositories, and more. These models have proven to be suitable for applications ranging from complex application modernization tasks (IBM, 2023) to on-device memory-constrained use cases. Extensive evaluation demonstrates that Granite Code models consistently reach state-of-the-art performance among open-source code LLMs, matching or exceeding the performance of recently released CodeGemma, StarCoder2, and Llama3 models on average performance across various code-related tasks of code generation, explanation, and bug fixing in a variety of popular programming languages. Our experience and results demonstrate that Granite Code models have a proven ability to better handle different tasks in enterprise software development workflows. We release all our Granite Code models under an Apache 2.0 license for both research and commercial use. We plan to continuously release updates to these models to improve their performance, e.g. leveraging the CodeNet instruction dataset (Puri et al., 2021), and in the near future we plan to release long-context as well as Python- and Java-specialized model variants.

Acknowledgments

We would like to acknowledge the efforts of numerous teams at IBM Research AI and Hybrid Cloud Platform, IBM AI Infrastructure team, IBM WatsonX Code Assistant and platform team. Special thanks to IBM Research leaders - Dario Gil, Sriram Raghavan, Mukesh Khare, Danny Barnett, Talia Gershon, Priya Nagpurkar, Nicholas Fuller for their support.

Thanks and acknowledgement to Trent Gray-Donald, Keri Olson, Alvin Tan, Hillery Hunter, Dakshi Agrawal, Xuan Liu, Mudhakar Srivatsa, Raghu Kiran Ganti, Carlos Costa, Darrell Reimer, Maja Vukovic, Dinesh Garg, Akash Srivastava, Abhishek Bhandwaldar, Aldo Pareja, Shiv Sudalairaj, Atin Sood, Sandeep Gopisetty, Nick Hill, Ray Rose, Tulio Coppola, Állysson Oliveira, Aadarsh Sahoo, Apoorve Mohan, Yuan Chi Chang, Jitendra Singh, Yuya Ong, Eric Butler, David Brotherton, Rakesh Mohan, David Kung, Dinesh Khandelwal, Naigang Wang, Nelson Mimura Gonzalez, Olivier Tardieu, Tuan Hoang Trong, Luis Angel Bathen, Kevin O'Connor, Christopher Laibinis, Tatsuhiro Chiba, Sunyanan Choochotkaew, Robert Walkup, Antoni Viros i Martin, Adnan Hoque, Davis Wertheimer and Marquita Ellis.

References

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*, 2021.

AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL CARD.md .

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

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.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. *arXiv preprint arXiv:2310.10631*, 2023.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

Kinjal Basu, Ibrahim Abdelaziz, Subhajit Chaudhury, Soham Dan, Maxwell Crouse, Asim Munawar, Sadhana Kumaravel, Vinod Muthusamy, Pavan Kapanipathi, and Luis A Lastras. Api-blend: A comprehensive corpora for training and benchmarking api llms. *arXiv preprint arXiv:2402.15491*, 2024.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle, 2022.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 2023.

Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. Can it edit? evaluating the ability of large language models to follow code editing instructions, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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, 2021.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

CodeGemma Team, Ale Jakse Hartman, Andrea Hu, Christopher A. Choquette-Choo, Heri Zhao, Jane Fine, Jeffrey Hui, Jingyue Shen, Joe Kelley, Joshua Howland, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Nam Nguyen, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Siqi Zuo, Tris Warkentin, and Zhitao et al. Gong. Codegemma: Open code models based on gemma. 2024. URL https://goo.gle/codegemma

Cohere. Command r+. https://docs.cohere.com/docs/command-r-plus

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf

Databricks. Introducing dbrx: A new state-of-the-art open llm — databricks blog. https://www.databricks.com/blog/introducing-dbrx-new-state-art-open-llm

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=wgDcbBMSfh

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36, 2024.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 12 2023. URL https://zenodo.org/records/10256836

Gemma-Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy.Gemma: Open models based on gemini research and technology, 2024.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021.

Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*, 2024.

IBM. watsonx code assistant, 2023. URL https://www.ibm.com/products/watsonx-code-assistant

Neel Jain, Ping yeh Chiang, Yuxin Wen, John Kirchenbauer, Hong-Min Chu, Gowthami Somepalli, Brian R. Bartoldson, Bhavya Kailkhura, Avi Schwarzschild, Aniruddha Saha, Micah Goldblum, Jonas Geiping, and Tom Goldstein. Neftune: Noisy embeddings improve instruction finetuning, 2023.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023a.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023b.

Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of bfloat16 for deep learning training, 2019.

Dahyun Kim, Chanjun Park, Sanghoon Kim, Wonsung Lee, Wonho Song, Yunsu Kim, Hyeonwoo Kim, Yungi Kim, Hyeonju Lee, Jihoo Kim, Changbae Ahn, Seonghoon Yang, Sukyung Lee, Hyunbyung Park, Gyoungjin Gim, Mikyoung Cha, Hwalsuk Lee, and Sunghun Kim. Solar 10.7b: Scaling large language models with simple yet effective depth up-scaling, 2024.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023. URL https://proceedings.mlsys.org/paper files/paper/2023/hash/e851ca7b43815718fbbac8afb2246bf8-Abstract-mlsys2023.html

Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, Shahul ES, Sameer Suri, David Glushkov, Arnav Dantuluri, Andrew Maguire, Christoph Schuhmann, Huu Nguyen, and Alexander Mattick. Openassistant conversations – democratizing large language model alignment, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Ariel N. Lee, Cole J. Hunter, and Nataniel Ruiz. Platypus: Quick, cheap, and powerful refinement of llms. 2023.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857, 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, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour 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!, 2023a.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023b.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023a. URL https://openreview.net/forum?id=1qvx610Cu7

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023b.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, et al. The flan collection: Designing data and methods for effective instruction tuning. In *International Conference on Machine Learning*, pp. 22631–22648. PMLR, 2023.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al.Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=r1gs9JgRZ

MistralAI. Mixtral 8x22b. https://mistral.ai/news/mixtral-8x22b/

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models, 2023.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209

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, 2023.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24. Association for Computing Machinery, 2024.

Keiran Paster, Marco Dos Santos, Zhangir Azerbayev, and Jimmy Ba. Openwebmath: An open dataset of high-quality mathematical web text. *arXiv preprint arXiv:2310.06786*, 2023.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, et al. Stable code technical report. *arXiv preprint arXiv:2404.01226*, 2024.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *NeurIPS*, 2021.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

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. *arXiv preprint arXiv:2308.12950*, 2023.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

Noam Shazeer. Glu variants improve transformer, 2020.

Yikang Shen, Zhen Guo, Tianle Cai, and Zengyi Qin. Jetmoe: Reaching llama2 performance with 0.1 m dollars. *arXiv preprint arXiv:2404.07413*, 2024.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Snowflake. Snowflake arctic - llm for enterprise ai. https://www.snowflake.com/blog/arctic-open-efficient-foundation-language-models-snowflake/

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu.Roformer: Enhanced transformer with rotary position embedding, 2023.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Shiqi Wang, Li Zheng, Haifeng Qian, Chenghao Yang, Zijian Wang, Varun Kumar, Mingyue Shang, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. Recode: Robustness evaluation of code generation models. 2022. doi: 10.48550/arXiv.2212.10264. URL https://arxiv.org/abs/2212.10264

Zhilin Wang, Yi Dong, Jiaqi Zeng, Virginia Adams, Makesh Narsimhan Sreedhar, Daniel Egert, Olivier Delalleau, Jane Polak Scowcroft, Neel Kant, Aidan Swope, and Oleksii Kuchaiev. Helpsteer: Multi-attribute helpfulness dataset for steerlm, 2023.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 10524–10533. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/xiong20b.html

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Berkeley function calling leaderboard. https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html, 2024.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*, 2023.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. Mammoth: Building math generalist models through hybrid instruction tuning. *arXiv preprint arXiv:2309.05653*, 2023.

Biao Zhang and Rico Sennrich. Root mean square layer normalization, 2019.

Yifan Zhang. Stackmathqa: A curated collection of 2 million mathematical questions and answers sourced from stack exchange, 2024.

A Programming Languages

ABAP, Ada, Agda, Alloy, ANTLR, AppleScript, Arduino, ASP, Assembly, Augeas, Awk, Batchfile, Bison, Bluespec, C, C-sharp, C++, Clojure, CMake, COBOL, CoffeeScript, Common-Lisp, CSS, Cucumber, Cuda, Cython, Dart, Dockerfile, Eagle, Elixir, Elm, Emacs-Lisp, Erlang, F-sharp, FORTRAN, GLSL, GO, Gradle, GraphQL, Groovy, Haskell, Haxe, HCL, HTML, Idris, Isabelle, Java, Java-Server-Pages, JavaScript, JSON, JSON5, JSONiq, JSONLD, JSX, Julia, Jupyter, Kotlin, Lean, Literate-Agda, Literate-CoffeeScript, Literate-Haskell, Lua, Makefile, Maple, Markdown, Mathematica, Matlab, Objective-C++, OCaml, OpenCL, Pascal, Perl, PHP, PowerShell, Prolog, Protocol-Buffer, Python, Python-traceback, R, Racket, RDoc, Restructuredtext, RHTML, RMarkdown, Ruby, Rust, SAS, Scala, Scheme, Shell, Smalltalk, Solidity, SPARQL, SQL, Stan, Standard-ML, Stata, Swift, SystemVerilog, Tcl, Tcsh, Tex, Thrift, Twig, TypeScript, Verilog, VHDL, Visual-Basic, Vue, Web-Ontology-Language, WebAssembly, XML, XSLT, Yacc, YAML, Zig