

MegaCode: A Repository-Level Dataset for Long-Context Code Reasoning

Wenyang Luo^{1*}, Shuang Sun^{1*}, Hu Yiwen^{1*}, Fei Bai¹, Yanzipeng Gao¹

Shuo Wang², Hongyu Lu³, Wayne Xin Zhao^{1✉}, Ji-Rong Wen¹

¹ Gaoling School of Artificial Intelligence, Renmin University of China

² Computer Science and Technology, Tsinghua University, Tsinghua University

³ Wechat, Tencent

wenyang_luo@outlook.com batmanfly@gmail.com

Abstract

Code reasoning is essential for building AI systems that can understand, generate, and verify complex software. Yet most open-source code corpora focus on isolated snippets or single functions, failing to capture the long-range dependencies and cross-file interactions that define real-world repositories. We introduce MegaCode, a 320B-token repository-level corpus comprising 310B source-code tokens and 10B repository-aware augmentation tokens. MegaCode preserves full project structure—including build scripts, configuration files, and inter-file links—and is constructed through a multi-stage curation pipeline combining deduplication, rule-based filtering, and model-based quality assessment. In controlled continued-pretraining experiments on a 10B-token subset, structured MegaCode improves the average code benchmark score from 20.25 with the same data flattened to 20.77, and the average RULER score from 73.31 to 74.46. Experiments on this subset also evaluate augmentations such as Language Server Protocol (LSP) metadata and repository summaries. We release the dataset, curation pipeline, metadata, and evaluation suite to facilitate research on large-scale code pretraining and repository-level understanding at <https://github.com/RUCAIBox/MegaCode>.

1 Introduction

Code large language models (LLMs) have achieved strong performance on function-level synthesis and understanding tasks, driven by advances in model architectures and increasingly large pretraining corpora. Recent models such as DeepSeek-Coder (Zhu et al., 2024) and Qwen2.5-Coder (Hui et al., 2024) demonstrate substantial gains on benchmarks like HumanEval (Chen et al., 2021), highlighting the effectiveness of scaling code-centric pretraining.

These improvements, however, are largely built on datasets that treat code as collections of single files or short snippets.

Most existing open-source code corpora—including The Stack v1 and v2 (Lozhkov et al., 2024), OpenCoder (Huang et al., 2025), and SwallowCode (Fujii et al., 2025)—focus on improving data quality through large-scale collection, deduplication, and filtering. While these efforts substantially reduce noise and redundancy, they largely flatten repository structure during preprocessing. As a result, critical information such as cross-file dependencies, build scripts, configuration files, and project-level organization is either discarded or treated independently, limiting the ability of pretrained models to learn long-range code dependencies.

To address these challenges, we introduce MegaCode, a 320-billion-token dataset comprising 310B source-code tokens and 10B repository-aware augmentation tokens. To ensure high-quality data at scale, we develop a multi-stage curation pipeline combining large-scale deduplication, rule-driven filtering, and model-based quality scoring. The pipeline is designed to preserve structural coherence while improving data cleanliness. We further augment the corpus with Language Server Protocol (LSP) metadata, repository structure, and repository summaries in controlled experiments.

We conduct continued-pretraining experiments on a 10B-token subset to evaluate repository-level training. A controlled comparison that holds the underlying data, token budget, model, and training recipe fixed shows higher code and long-context averages when repository structure is preserved rather than flattened.

Our contributions are threefold:

- A 320B-token, repository-level, multilingual corpus for long-context code reasoning, with explicit accounting for 310B source-code tokens and 10B augmentation tokens.

* Equal contribution.

✉ Corresponding author.

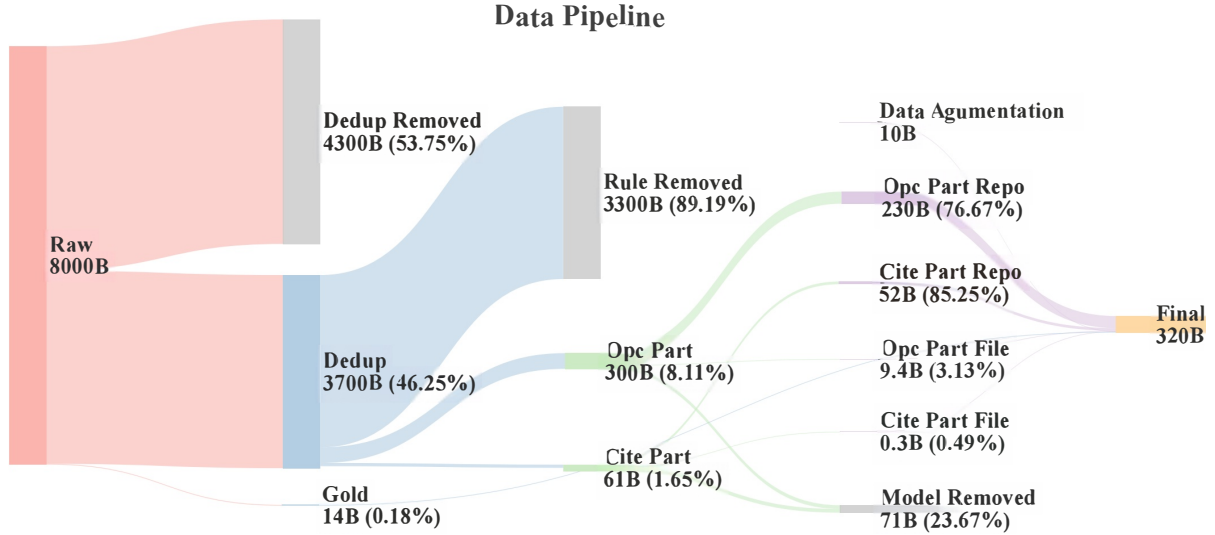


Figure 1: End-to-end data pipeline for constructing MegaCode. Pre-filtering (not shown) produces MegaCode-raw with 8T tokens. Exact deduplication removes 53.75% of these tokens while preserving essential environment files as the Gold branch. OpenCoder-derived rule filtering produces the OPC branch, and citation-connectivity reintegration produces the Cite branch; model-based filtering is then applied as shown. The resulting 310B source-code tokens are supplemented with 10B LSP- and summary-augmentation tokens, yielding the released 320B-token corpus.

- A fully open curation pipeline that integrates repository preservation, deduplication, multi-stage filtering, and data augmentation.
- Controlled evaluation on a 10B-token subset, including a same-data structured-versus-flattened ablation and an analysis of LSP metadata and repository summaries.

We release the dataset and full pipeline to support open research on large-context code pretraining and to encourage future work on models capable of reasoning over full software repositories.

2 MegaCode Construction

This section describes the construction of MegaCode, including data sources, preprocessing, multi-stage filtering, quality scoring, augmentation, and final formatting. Our goal is to build a large-scale repository-level corpus that preserves real-world software structure while ensuring high data quality suitable for long-context pretraining. The overall data-cleaning pipeline is shown in Figure 1.

2.1 Data Source

Existing open-source corpora—such as The Stack v2 (Lozhkov et al., 2024)—primarily contain source code collected up to September 2023, and therefore only partially reflect recent shifts in software development practices, tooling ecosystems, and programming language usage. To address this temporal limitation and improve coverage of con-

temporary code patterns, we construct an additional corpus by crawling GitHub repositories created between April 2023 and June 2025. This design choice avoids overlap with existing datasets such as SwallowCode (Fujii et al., 2025), which is built from The Stack v2.

2.1.1 Pre-Filtering

We first download repository snapshots in compressed form. During decompression, we perform lightweight pre-filtering to remove noisy or invalid content. Binary files (e.g., media, archives, executables, compiled artifacts, and malware), files exceeding a 10MB size threshold, and repositories whose snapshots fail to unzip are discarded. Only text-based source artifacts are retained. We also remove common shared templates that would create extensive duplication in later stages, such as `node_modules` and Python environment directories. This forms our MegaCode-raw dataset.

2.2 Process Pipeline

MegaCode is constructed through a multi-stage pipeline designed to (1) preserve repository structure, (2) remove redundancy, and (3) improve quality without discarding structurally important components based on MegaCode-raw.

2.2.1 Deduplication

We conduct large-scale deduplication to improve data quality while preserving the natural diversity

of software development. The process combines exact hashing with controlled retention of structurally meaningful duplicates.

- *Exact deduplication.* We apply SHA-256 hashing to detect fully identical files within and across repositories. Most duplicated files are shared libraries, autogenerated artifacts, and shared templates. Because these files can play functional roles in build pipelines or project layouts, we retain the content in the highest-star repository to keep that repository as complete as possible and retain only the filenames in other repositories. This preserves structural relationships without repeatedly including identical content.

- *Preservation of near-duplicates.* Unlike natural-language corpora, code repositories frequently contain incremental revisions of the same module. These near-duplicate variants capture meaningful evolution, including refactoring steps and version-specific updates. Aggressive fuzzy deduplication would remove these useful distinctions and reduce representational diversity. Therefore, we intentionally retain near-duplicates to support learning of software evolution patterns and fine-grained structural variations.

2.2.2 Rule-based Filtering

After deduplication, we apply a series of rule-based filters informed by both OpenCoder (Huang et al., 2025) and our own empirical findings.

We first remove extremely long files (>128 kB), which account for only approximately 1% of the raw corpus yet dominate processing time and are frequently correlated with autogenerated or low-value content such as logs. We then apply OpenCoder’s quality signals to the remaining files, including the number of lines, the presence of long lines, and repeated n -gram counts. Thresholds on these signals determine whether a file is marked for removal. However, these rules can be overly strict and occasionally discard files that play important roles within a repository.

To better preserve structural integrity, we reintroduce files that exhibit high citation connectivity. Specifically, we treat the first 1,000 characters of each file as its reference zone. If another file’s base name appears within this region, we record an outgoing citation; incoming citations are computed analogously. Using these statistics, we identify the number of files a document cites and the number of files that cite it. Files with both counts exceeding two are deemed structurally important and are

preferentially retained.

2.2.3 Model-Based Filtering

Beyond rule-based filtering, we incorporate a model-based quality assessment at both the file and repository levels. Repositories containing more than five files are evaluated using repository-level criteria, while smaller repositories are assessed at the file level. To comprehensively evaluate data quality and maximize information extraction, we annotate four key aspects: summary, key details, weaknesses, and rating. The definitions and evaluation criteria for these aspects differ between the file-level and repository-level assessments.

File-level Scoring. For repositories containing fewer than five files, we assume that the structural information available at the repository level is relatively limited. Therefore, such repositories are decomposed and annotated at the file level. The four annotation aspects are as follows:

- *Summary:* An overview of the file’s overall functionality, technology stack, workflow *etc.*
- *Key details:* Extraction of key implementation strategies and highlights of the core logic.
- *Weakness:* Identification of issues related to structure, performance, or coding practices.
- *Rating:* A quality score ranging from 1 to 10.

We reuse the evaluation criteria of Seed Coder (Seed et al., 2025) as shown in Appendix A. We use Qwen3-32B (Qwen-Team, 2025) to annotate approximately 700,000 instances and fine-tune Qwen3-4B-Instruct-2507 (Qwen-Team, 2025) to obtain the final file-level quality evaluator.

Repository-level Scoring. For repositories containing more than five files, we retain the complete repository structure and conduct annotation and filtering at the repository level. The four annotation aspects at this level are:

- *Summary:* A concise description that explains the repository’s core features, functionality, employed technologies, and overall purpose.
- *Key details:* Extraction of concrete implementation details that reflect the repository’s architecture and design decisions.
- *Weaknesses:* Identification of potential weaknesses, limitations, or areas for improvement within the repository.
- *Rating:* A numerical quality score assigned to the repository on a scale from 1 to 10.

Because repositories vary greatly in size, we further categorize them based on their scale. For



```

1  # <--symbols information about this file-->
2  | Variable: app
3  | Function: home
4  | Function: post_data
5  |   Variable: data
6  |   Variable: model
7
8  def app():
9      # ...
10     pass
11
12 def home():
13     '''Render the home page'''
14     l = ['MedInc', 'HouseAge', 'AveRooms']
15
16     #<--hovers about render_template function-->
17     (function) def render_template(
18         template_name_or_list: Text | Iterable[Text],
19         **context: Any
20     ) -> Text
21     Render a template by name with the given context.
22     :param template_name_or_list: The name of the
23     template to render. If a list is given,
24     the first name to exist will be rendered.
25     :param context: The variables to make available
26     in the template.
27
28     #<--function call of render_template-->
29     return render_template("index.html", labels=l)
30
31 #<--diagnostics information about this file-->
32 warning: post_data function is defined but not used

```

Figure 2: Demonstration of Language Server Protocol (LSP): symbols summarize functions and variables, hover shows a function’s signature and documentation, function calls mark invocation sites, and diagnostics report static warnings such as unused definitions.

small repositories, we provide full annotations for all four aspects. For large repositories, however, the context-length constraints of the model—and the need to avoid hallucinations due to incomplete information—lead us to annotate only the summary and the rating. In addition, we implement several optimizations during scoring:

- *Code unformatting*: For Python, C++, Java, and C#, partial removal of formatting does not affect program logic (Pan et al., 2025). Therefore, we strip formatting from these source files to further compress the input context.

- *File ordering*: Based on the previously annotated file citation information, we sort the files within each repository and prioritize higher-ranked files during annotation.

Using Qwen3-32B, we annotate approximately 200,000 repository-level instances. These annotations are then used to train Qwen3-4B-Instruct-2507, which serves as our final repository-level evaluator. An example repository annotation is shown in Appendix A.

We validate the distilled Qwen3-4B scorer on 100 held-out Qwen3-32B annotations, obtaining a mean squared error (MSE) of 1.94. We also manually inspect approximately 1,000 scored samples and find that the assigned scores are largely consistent with our judgments. The latter is a qualitative sanity check rather than an independent human-annotation study.

2.3 Data Augmentation

To enhance the structural richness of high-quality repositories, we apply program-analysis-based augmentation centered on two forms of semantic extraction. A demo is shown in Figure 2.

- *Fine-grained function call extraction*. For each supported language (e.g., Python, C/C++, Go), we construct fine-grained call relations by parsing the Abstract Syntax Tree (AST). The AST enables precise identification of call sites, callee names, and contextual information such as enclosing scopes or control structures. To enrich these call edges with semantic detail, we query Language Server Protocol (LSP) services for each callee’s hover information, obtaining function signatures, type hints, documentation snippets, and definition locations. This combination yields a detailed, cross-file call graph with high semantic fidelity.

- *File-level symbol extraction*. Beyond call relationships, we also aim to capture the complete structural layout of individual files. Using LSP symbol queries, we extract all top-level and nested symbols—functions, classes, methods, constants, and module-level definitions—to form a unified structural summary for each file. This symbol inventory provides comprehensive insight into the file’s internal organization and supports global reasoning over repository-scale codebases.

Together, LSP-derived metadata and model-generated summaries contribute 10B augmentation tokens. Combined with the 310B filtered source-code tokens, the released corpus contains 320B tokens in total.

2.4 Data Formatting

After filtering, we convert all retained files into either file-level or repo-level format. Examples are shown in Appendix B.

2.4.1 File-Level Formatting

Each file is normalized independently before being assembled into repository-level examples. We implement several formatting strategies:

- *LSP format.* For languages with LSP metadata, we enrich each file with symbolic and semantic information obtained from hover queries, symbol-tree extraction, and diagnostic reports. Symbol trees are constructed into a hierarchical view of classes, functions, and methods, which is wrapped with `<|ignore_start|>` and `<|ignore_end|>`. Hover annotations provide signatures, type hints, documentation snippets, and definition locations; these are injected at the corresponding source lines and wrapped with `<|keep_start|>` and `<|keep_end|>`. Diagnostics—including compiler or LSP warnings—are appended as extra information and wrapped with `<|keep_start|>` and `<|keep_end|>`.

- *FIM format.* When LSP metadata is unavailable, we assign a file to FIM or normal format with equal probability. In FIM format, files are split into prefix, middle, and suffix segments using a stochastic line-aware splitting algorithm that prefers semantically meaningful boundaries. The segments are then wrapped with `<|fim_prefix|>`, `<|fim_middle|>`, and `<|fim_suffix|>` tokens.

- *Normal format.* Files not assigned to the LSP or FIM pathways retain their original content. For files marked for removal in the preceding stages, we replace the content with “`<|ignore_start|>` This file was ignored. `<|ignore_end|>`”.

As described in Section 2.2.3, we divide the data into file-level and repository-level examples. In the file-level format, each file is preceded by its summary, a `<|file_sep|>` marker, and its path to align with the repository format in Section 2.4.2.

2.4.2 Repository-Level Formatting

Files following the conditions of Section 2.2.3 will be assembled into coherent long-context units. A repository begins with metadata fields wrapped by control tokens such as `<|repo_name|>` and repository name, then an `<|ignore|>` block containing a textual directory tree and optional repository summary information.

Files are then concatenated in a deterministic traversal order. Each file is preceded by a `<|file_sep|>` marker and its full repository-relative path, enabling the model to reason about cross-file structure. The final repository-level example therefore contains: a machine-readable summary of repository geometry (file tree), optional repository-level explanations, and the sequence of file-level formatted texts.

This integrated layout provides consistent struc-

tural priors while allowing the model to consume complete multi-file contexts during training.

2.5 Dataset Summary and Comparison

Stage	# Files	# Repos	Tokens (B)
Raw	1.38B	23.38M	8000
Dedup	0.52B	21.61M	3700
OPC	300M	18.83M	300
Cite	57.34M	6.24M	61
Gold	23.76M	9.17M	14
Final source code	311.9M	16.7M	310
Augmentation	—	—	10
Released total	—	—	320

Table 1: Dataset statistics and token accounting. OPC denotes files retained by the OpenCoder-derived rules, Cite denotes files reintroduced based on citation connectivity, and Gold denotes essential environment files preserved during deduplication. The final 310B source-code tokens plus 10B augmentation tokens yield the released 320B-token corpus. Intermediate branches are not additive because model-based filtering follows rule filtering and reintegration.

Figure 1 and Table 1 summarize the MegaCode curation process. While the total number of files is reduced by over $4\times$ (1.38B to 311.9M), the number of repositories is largely preserved (23.38M to 16.7M). In contrast, the source-code token count drops sharply from 8T to 310B, indicating that the pipeline primarily removes excessively long, low-quality, or redundant content rather than aggressively pruning repositories. Adding 10B augmentation tokens gives the released total of 320B tokens. This design prioritizes repository integrity over file-level retention to expose models to long-range code dependencies.

Table 2 situates MegaCode relative to existing large-scale code datasets. Prior corpora such as The Stack v2 (Lozhkov et al., 2024) and OpenCoder (Huang et al., 2025) treat files largely independently and do not incorporate semantic augmentation beyond filtering. SwallowCode (Fujii et al., 2025) applies deeper transformations but is limited to a single language and smaller scale. More recent datasets, such as Nemotron CC Code v1 (Blakeman et al., 2025), combine rule-based and model-based filtering, yet still operate primarily at the file level.

In contrast, MegaCode simultaneously provides an open-source pipeline and cleaned data, supports multilingual repositories at scale, and augments raw code with repository-aware signals such as LSP-derived annotations and summaries. These augmentations expose models to explicit structural

Dataset	Pipe	Data	Tokens	Multilingual	Rule Filter	Model Filter
The Stack v2 (Lozhkov et al., 2024)	Yes	Yes	900B	Yes	Yes	No
OpenCoder (Huang et al., 2025)	Yes	Yes	960B	Yes	Yes	No
SwallowCode (Fujii et al., 2025)	Yes	Yes	16B	No	Yes	Yes
SeedCoder (Seed et al., 2025)	No	No	-	Yes	Yes	Yes
Nemotron CC Code v1 (Blakeman et al., 2025)	No	Yes	427B	Yes	Yes	Yes
MegaCode (Ours)	Yes	Yes	320B	Yes	Yes	Yes

Table 2: Comparison of large-scale code datasets in terms of processing pipeline, scale, and structural properties. Pipe and Data indicate whether the data-cleaning code and pretraining data are open, respectively. Tokens denotes corpus size; the MegaCode total includes 310B source-code tokens and 10B augmentation tokens. Rule Filter and Model Filter indicate the use of manual rules and model-based scores.

and semantic information—symbol definitions, references, and project-level intent—that is implicit or absent in previous datasets. Given the development of code agents, MegaCode could serve as a valuable foundation for pretraining models to better leverage LSP tool capabilities. Additionally, its long-context, repository-level training may benefit future software engineering projects.

3 Experiments

In this section, we provide a detailed overview of our training experiments to validate our dataset.

3.1 Training Setup

Model Architecture Because long-context pretraining is computationally expensive, we conduct continued pretraining with the long-context Llama-3.2-1B model (Grattafiori et al., 2024). The model architecture is shown in Table 5.

Data Setting For the selected model, we perform continued pretraining on a total of 10B tokens under multiple data configurations. Before training, we apply benchmark-specific 10-gram decontamination against EvalPlus, MultiPL-E, CRUXEval, and RULER; this removes 0.56% of the training corpus. To better align the data distribution with downstream coding tasks, we downsample HTML-related content in MegaCode to 50% of its original volume and truncate CSV and JSONL files to 2k tokens. We convert Jupyter notebooks into standard Python files using scripts from Yiwu et al. (2025). To prevent leakage of personally identifiable information (PII), we use the StarPII model (Lozhkov et al., 2024) to detect and redact PII entities.

To evaluate the data augmentations introduced in Section 2.3, we consider variants that incorporate (i) LSP-derived structural signals, (ii) natural language summaries, and (iii) their combination.

We also include a configuration in which LSP signals and summaries are appended to the end of the sequence. To isolate repository formatting, we compare structured and flattened versions of the same MegaCode data while holding the token budget, model, and training recipe fixed.

We additionally train on SwallowCode (Fujii et al., 2025) with the same 10B-token budget and hyperparameters. We select it as a strong filtered corpus derived from The Stack v2; OpenCoder’s training corpus is not publicly available for a direct training comparison. As SwallowCode is Python-centric whereas MegaCode is multilingual, this baseline illustrates a data-composition trade-off.

To preserve the model’s instruction-following capability during continued pretraining, we mix 9.5B tokens of code data with 0.5B tokens sampled from Infinity Instruct (Li et al., 2025) in all settings.

Training Details For all experiments, we use a cosine learning-rate scheduler with a peak learning rate of 2×10^{-5} and a minimum learning rate of 1×10^{-7} , with a warm-up phase of 1B tokens. The global batch size is 128 and the sequence length is 32,768. Training is conducted using Megatron-LM (Shoeybi et al., 2019) with distributed optimization and DDP gradient overlap on a cluster of 32 H20 GPUs over a total of 13.2 hours, equating to 422.4 GPU hours.

3.2 Evaluation Setup

Benchmarks and Metrics We follow the qwen-coder-eval framework¹ and evaluate on EvalPlus (Liu et al., 2023), which includes HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), HumanEval+, and MBPP+; MultiPL-E (Cassano et al., 2023) in Java, C++, JavaScript, C#, PHP, Bash, and TypeScript; and the 2-shot

¹<https://github.com/QwenLM/Qwen3-Coder>

CRUX-I and CRUX-O tasks from CRUXEval (Gu et al., 2024).

To evaluate long-context processing, we use RULER (Hsieh et al., 2024) with the OpenCompass framework (OpenCompass, 2023). RULER includes tasks such as needle-in-a-haystack retrieval, multi-step reasoning, and mathematical reasoning.

All results are reported using pass@1 accuracy without enabling thinking mode. For code generation tasks, we use greedy decoding with a maximum generation length of 1024 tokens. Each training configuration is run once; thus, the reported comparisons are descriptive and do not support significance claims or a definitive ranking among closely performing variants.

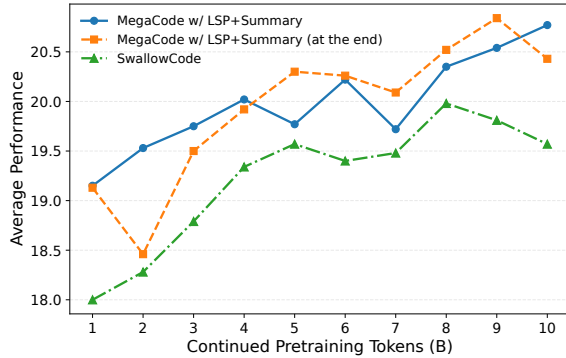


Figure 3: Average coding benchmark performance throughout continued pretraining from 1B to 10B tokens. “LSP” denotes LSP-derived structural signals, “Summary” denotes natural-language code summaries, and “LSP+Summary” uses both augmentations. “LSP+Summary (end)” denotes the variant in which the LSP signals and summaries are appended to the end of the input sequence. Each point reports the average pass@1 score across EvalPlus, MultiPL-E, and CRUXEval at the corresponding training checkpoint. Detailed results are shown in Table 6.

3.3 Results Analysis

Average scores under different settings are shown in Figure 3 and Table 3, with detailed results in Table 6. The structured-versus-flattened ablation and RULER results are shown in Table 4.

MegaCode improves the code average on the tested 10B-token subset As shown in Figure 3 and Table 3, the base model obtains an overall average of 19.33, while the four tested MegaCode augmentation configurations obtain 20.13–20.77 after 10B tokens of continued pretraining. The LSP+Summary configuration reaches 20.77, a relative improvement of approximately 7.4% over the

base model. These results are scoped to one 1B-parameter model, one seed, and the tested subset.

Under the same 10B-token training budget, MegaCode with LSP+Summary yields a higher overall average than SwallowCode (20.77 vs. 19.57). SwallowCode is competitive or stronger on several Python-centric measures, including HumanEval+ (19.50 vs. 17.70) and MBPP+ (34.80 vs. 32.60), whereas multilingual MegaCode is stronger on examples such as MultiPL-E JavaScript (18.63 vs. 14.29), Java (15.19 vs. 10.13), and TypeScript (20.75 vs. 16.35). Because SwallowCode is Python-centric, these differences primarily demonstrate the benefit of MegaCode’s multilingual composition rather than superior curation quality.

LSP signals and natural language summaries enhance model performance Table 3 shows that the runs with either LSP-derived signals or natural language summaries obtain overall averages of 20.13 and 20.35, respectively, compared with 19.33 for the base model. Combining both obtains 20.77 in this run. The augmentation data account for only approximately 1/32 of the training tokens, and the differences among MegaCode variants are small relative to the checkpoint fluctuations in Figure 3. We therefore treat these results as evidence that such signals can be useful, not as establishing a definitive ranking among variants.

Appending LSP signals and summaries at the end of the input sequence obtains an average of 20.43. This observation suggests that the model can use auxiliary structural information in this placement, although confirming the effect requires repeated runs. Because LSP-style signals are also used by code agents, pretraining with such annotations may provide a useful foundation for tool-aware models.

Preserving repository structure improves the controlled ablation Table 4 isolates formatting by using the same MegaCode data and training setup for the structured and flattened variants. Preserving structure raises the code average from 20.25 to 20.77 and the RULER average from 73.31 to 74.46. The flattened variant is slightly stronger at 4k (84.40 vs. 83.73), and the variants tie at 8k (76.97); the structured variant is stronger at 16k (73.28 vs. 71.58) and 32k (63.85 vs. 60.29). Thus, the clearest benefit of repository organization in this experiment appears at longer contexts.

Model	EvalPlus				MultiPL-E							CRUX-Eval		AVG
	HE	HE+	MBPP	MBPP+	Java	C++	JS	C#	PHP	Shell	TS	Input	Output	
Llama-3.2-1B	18.30	15.90	39.10	32.60	13.92	15.53	15.53	17.09	9.94	1.90	18.24	29.00	24.20	19.33
+MegaCode 10B														
w/ LSP+Summary	22.00	17.70	41.10	32.60	15.19	17.39	18.63	16.46	10.56	2.53	20.75	31.90	23.20	20.77
w/ LSP	20.70	16.50	41.10	33.30	14.56	15.53	16.77	18.99	10.56	2.53	17.61	29.90	23.60	20.13
w/ Summary	23.80	17.10	39.80	31.80	15.19	15.53	16.77	18.99	12.42	2.53	18.87	29.80	22.00	20.35
w/ LSP+Summary (end)	20.70	17.10	40.10	33.60	15.19	18.63	17.39	15.82	13.04	1.90	16.98	32.20	22.90	20.43
+SwallowCode 10B														
SwallowCode	22.60	19.50	41.10	34.80	10.13	12.42	14.29	18.35	11.80	1.90	16.35	29.60	21.60	19.57

Table 3: Overall code benchmark performance under different continued pretraining data configurations. Results are averaged across EvalPlus, MultiPL-E, and CRUXEval benchmarks.

Training data/format	Code Avg	RULER 4k	8k	16k	32k	RULER Avg
Llama-3.2-1B (base)	19.33	78.20	71.89	65.32	60.29	68.93
SwallowCode	19.57	80.11	71.05	68.36	56.48	69.00
MegaCode (flattened)	20.25	84.40	76.97	71.58	60.29	73.31
MegaCode (structured)	20.77	83.73	76.97	73.28	63.85	74.46

Table 4: Controlled structured-versus-flattened ablation and RULER results. The two MegaCode rows use identical underlying data, token budget, model, and training recipe; they differ only in whether repository organization is preserved. Code Avg averages EvalPlus, MultiPL-E, and CRUXEval. Base and SwallowCode results are included for context but are not part of the same-data formatting ablation.

MegaCode yields substantial gains in long-context evaluation On RULER, structured MegaCode obtains an average of 74.46, compared with 68.93 for the base model and 69.00 for the SwallowCode-trained model. At 32k, it obtains 63.85, compared with 60.29 and 56.48, respectively. Together with the same-data ablation above, these results provide evidence that repository-preserving training benefits long-context processing under the tested 10B-token setting. We use “substantial” descriptively; multiple seeds are needed to determine statistical significance.

4 Related Work

Code Large Language Models The past two years have witnessed rapid advances in large language models explicitly designed for code understanding and generation. Code LLMs such as DeepSeek-Coder (Zhu et al., 2024) and Qwen2.5-Coder (Hui et al., 2024) achieve substantial improvements on code benchmarks such as HumanEval (Chen et al., 2021). However, inaccessible pretraining data hinder further innovation. The Stack v1 and v2 (Lozhkov et al., 2024) open-source permissively licensed code from public repositories; their use in the StarCoder models demonstrates the utility of these corpora. OpenCoder (Huang et al., 2025) and SwallowCode (Fu-

jii et al., 2025) further curate code data and show competitive results on several benchmarks.

Code Pretraining Data With the progress of code LLMs, constructing high-quality pretraining data has become increasingly important. The Stack v2 (Lozhkov et al., 2024) represents a major step in this direction by assembling a large, license-aware corpus from the Software Heritage archive and complementary sources, covering hundreds of programming languages. Beyond raw collection, OpenCoder (Huang et al., 2025) proposes a curated pipeline with more than 100 manually written rules that substantially reduce noise and redundancy. SwallowCode (Fujii et al., 2025) focuses on Python and studies the benefits of rewriting raw code into more standardized formats. In addition, Nemotron CC Code v1 (Blakeman et al., 2025) augments real-world code with synthetic transformations, reflecting growing interest in combining organic and generated data. These components are individually related to prior work; MegaCode’s primary contribution is their integration into an open, multilingual, repository-preserving corpus and construction pipeline, together with repository-aware LSP metadata and summaries.

Code Benchmarks and Evaluation The development of increasingly capable Code LLMs has driven parallel progress in evaluation methodolo-

gies. EvalPlus (Liu et al., 2023) strengthens classic function-level benchmarks by augmenting HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) with substantially larger and more diverse test suites, resulting in HumanEval+ and MBPP+, and thereby mitigating the issue of insufficient test coverage. MultiPL-E (Cassano et al., 2023) extends established code generation benchmarks to a multilingual setting by systematically translating tasks into multiple programming languages, including Java, C++, JavaScript, C#, PHP, Bash, and TypeScript, enabling controlled analysis of cross-language generalization. Complementary to synthesis-focused evaluations, CRUXEval (Gu et al., 2024) targets execution-level program understanding through CRUX-I and CRUX-O, which assess models’ ability to reason about program inputs and outputs rather than merely generating code.

5 Conclusion

We present MegaCode, a large-scale repository-structured code dataset designed for long-context, repository-level code reasoning. Unlike existing code corpora that primarily treat code as isolated files or snippets, MegaCode preserves global repository structure and enriches code with explicit structural signals, including LSP-derived annotations and summaries. Continued pretraining of Llama-3.2-1B on a 10B-token subset improves code and long-context evaluation relative to the base model. Moreover, a same-data controlled ablation shows that structured formatting improves the code average and RULER average over flattened formatting, with the largest RULER gains at 16k–32k context lengths. These results provide initial evidence for the value of repository-level organization while remaining scoped to the tested model and subset. We release MegaCode, together with its curation pipeline, metadata, and evaluation framework, to support future research on scalable, structure-aware code pretraining.

Limitations

Our study has several limitations. First, all experiments use continued pretraining of a 1B-parameter model, and the observed gains may not transfer directly to larger architectures. Second, although the released corpus contains 320B tokens (310B source-code tokens and 10B augmentation tokens), our empirical validation uses only a 10B-token subset; it therefore does not establish how performance

scales to the full corpus. Third, each configuration is trained once, so the reported differences lack uncertainty estimates and significance tests and should not be used to establish a definitive ranking among the augmentation variants. Finally, we do not report a downstream filtering ablation or residual benchmark-overlap analysis, nor do we explore the optimal granularity, formatting, or interaction of the LSP-derived annotations and summaries.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China under Grant No. 92470205, Beijing Natural Science Foundation under Grant No. L2611017, and Beijing Major Science and Technology Project under Contract No. Z251100008425002.

References

- Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.
- Aaron Blakeman, Aaron Grattafiori, Aarti Basant, Abhibha Gupta, Abhinav Khattar, Adi Renduchintala, Aditya Vavre, Akanksha Shukla, Akhiad Bercovich, Aleksander Ficek, and 1 others. 2025. Nvidia nemotron 3: Efficient and open intelligence. *arXiv preprint arXiv:2512.20856*.
- Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Trans. Software Eng.*, 49(7):3675–3691.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.
- Kazuki Fujii, Yukito Tajima, Sakae Mizuki, Hinari Shimada, Taihei Shiotani, Koshiro Saito, Masanari Ohi, Masaki Kawamura, Taishi Nakamura, Takumi Okamoto, and 1 others. 2025. Rewriting pre-training data boosts llm performance in math and code. *arXiv preprint arXiv:2505.02881*.
- Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten,

- Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. CRUXEval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.
- Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekish, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What’s the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*.
- Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Xianzhen Luo, Qiufeng Wang, Yuan-Tao Fan, Qingfu Zhu, Zhaoxiang Zhang, Yang Gao, Jie Fu, and 7 others. 2025. [OpenCoder: The open cookbook for top-tier code large language models](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33167–33193, Vienna, Austria. Association for Computational Linguistics.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Jijie Li, Li Du, Hanyu Zhao, Bo-wen Zhang, Liangdong Wang, Boyan Gao, Guang Liu, and Yonghua Lin. 2025. Infinity instruct: Scaling instruction selection and synthesis to enhance language models. *arXiv preprint arXiv:2506.11116*.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *NeurIPS*.
- Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.
- OpenCompass. 2023. Opencompass: A universal evaluation platform for foundation models. <https://github.com/open-compass/opencompass>.
- Dangfeng Pan, Zhensu Sun, Cenyuan Zhang, David Lo, and Xiaoning Du. 2025. The hidden cost of readability: How code formatting silently consumes your llm budget. *arXiv preprint arXiv:2508.13666*.
- Qwen-Team. 2025. [Qwen3 technical report](#). Preprint, arXiv:2505.09388.
- ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, and 1 others. 2025. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*.
- Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.
- Hu Yiwen, Huatong Song, Jie Chen, Jia Deng, Jiapeng Wang, Kun Zhou, Yutao Zhu, Jinhao Jiang, Zican Dong, Yang Lu, Xu Miao, Wayne Xin Zhao, and Jirong Wen. 2025. [YuLan-mini: Pushing the limits of open data-efficient language model](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5374–5400, Vienna, Austria. Association for Computational Linguistics.
- Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Prompts and results for model-based filtering

Prompt for File-level Annotation

You are an expert of coding. Please carefully evaluate the quality of the {LANGUAGE} code file below based on the specific quality criteria essential for its potential use in pretraining a large language model.

Your response must be structured strictly in **YAML format** with the following keys:

1. **Summary**: Provide a brief summary that explains the **overall functionality or purpose**, **technologies used**, and **general pipeline or logic** of the code.
2. **KeyDetails**: Focus on specific implementation details that are critical to the functionality, highlighting special handling, rules, conditions, or in-function logic, without repeating the simple overall logic or pipeline. Include examples such as different input/output processing under certain conditions, special masking, filtering or transformation rules, notable function calls or configurations, and other subtle implementation tricks. **KeyDetails** must be concrete and explicit, showing the actual implementation rather than just a general summary.
 - If the code does not contain important or high-quality details (e.g., it is too simple, trivial, or lacks meaningful internal logic), then set this field to **"No KeyDetails"**.
3. **Weaknesses**: Explicitly list the weaknesses or bad practices of the code, including why they are problematic. Examples: poor readability due to lack of comments, overly long functions without modularization, excessive hard-coded values, lack of error handling, or poor naming conventions.
4. **Rating**: A numerical rating to the code on a scale from 1 to 10, where 1 indicates the lowest quality and 10 indicates the highest quality. Note that the zero score policy should be firstly considered in your analysis, and skip the other criteria if the code meets any zero score conditions.

Do not include any explanation or commentary in the output, only the YAML with the four fields.

KeyDetails may cover (if present in the code):

- Core Logic: key algorithmic steps or special per-case handling inside functions, not the overall pipeline.
- Performance/Parallelism: concurrency strategy, parallel processing of data, and potential bottlenecks.
- Data Access and I/O Optimization: special reading, streaming, filtering, or caching techniques.
- Error Handling and Edge Cases: retries, exceptions, or fallbacks for corner cases.
- Memory and Serialization: in-memory processing, masking, truncation, or streaming of large datasets.
- API Style and Composability: modularity of individual functions, reusability, or encapsulation of critical logic.
- Other Details (optional): any clever hacks, unusual logic, or special configuration critical for functionality.

Criteria for Evaluation:

* Readability:

- Presence of a reasonable amount of comments.
- Inclusion of classes or functions, better with reasonable docstrings that describe the functionality.
- Neat and consistent formatting that adheres to common practice.
- Good naming conventions and well-structured code.

* Modularity:

- Avoidance of overly complicated / very long functions through modularization.
- Clear separation of logic and functionality, using classes and functions.
- Design of each module or component to perform a clear and coherent task.

* Clarity:

- Minimization of excessively repeated code and code blocks, such as repeatedly calling the same function for many times.
- Avoidance of massive commented-out code blocks.
- Avoidance of many random printing statements for debugging.
- Clear communication of intentions behind code blocks.

- * Reusability:
 - Absence of syntax or logical errors.
 - Avoidance of embedding lots of hard-coded data directly within the code.
 - Provision of complete and meaningful functionality, not overly simplistic.
 - Design that facilitates easy reuse of functions or classes in other projects.
- * Zero Score Policy:
 - If the code is mostly configurations, such as very long json objects with many numbers or strings, rate 0 score.
 - If the code is essentially a data file which includes lots of hard-coded data, such as too many lines of numbers or strings, rate 0 score.
 - If the code has little to none effective logic, or is dominated by literals or assignments without any complexity, rate 0 score.
 - If the code is auto-generated, with any comments like generated by Django, rate 0 score.

After reviewing the provided code, generate a brief summary, extract the key implementation details (only the ones that are relevant in the code or "No KeyDetails"), list weaknesses explicitly, and finally conclude with the rating based on the given criteria. Your output must strictly follow the YAML format.

Example Output:

```

```yaml
Summary: A Python script for fine-tuning a causal language model using HuggingFace
 Transformers, supporting custom tokenization, data processing. The script loads JSON
 datasets, tokenizes input/output pairs with specific masking rules, and trains the model
 using the Trainer API.
KeyDetails:
- |
 Core Logic: Iterates over sample["output"] for each sample, and for each key-value pair,
 if the value is not None, tokenizes the text. If the key is "gen", it appends the tokens
 to both input_ids and labels so they are included in loss computation. If the key is "
 doc_gen", it appends the tokens to input_ids but appends IGNORE_INDEX to labels to mask
 them from loss calculation, ensuring that in multi-turn dialogues only the prompt part is
 masked while generation outputs are appropriately handled for SFT training.
- |
 Model Loading & Performance: Loads the model with `AutoModelForCausalLM.from_pretrained
 (... , _attn_implementation="flash_attention_2")` to enable flash_attention_2, improving
 efficiency of attention computation during training, especially for long sequence inputs.
- |
 Memory and Serialization: Uses map() to process datasets in memory, filtering out
 sequences longer than 10,000 tokens to prevent overly long inputs.
- |
 Data Access and I/O Optimization: Loads JSON datasets using load_dataset('json'), supports
 merging multiple dataset types, processes each dataset with num_proc=1, and applies
 different masking strategies: special masking for standard datasets and standard SFT
 masking for other-type datasets.
- |
 API Style and Composability: Uses Trainer, DataCollatorForSeq2Seq, and TrainingArguments
 for modular training. Model and tokenizer arguments are encapsulated in dataclasses for
 clarity and reusability.
- |
 Error Handling and Edge Cases: Skips None values in output, ensures pad_token is set to
 eos_token if missing, and prevents gradient_checkpointing conflicts with use_cache.
Weaknesses:
- Converts dataset to a Python list, reducing efficiency and scalability.
- Lacks error handling and docstrings; minimal inline comments.
- Unused imports and a typo (`dailog_augmentation`).
Rating: 8
```

```

{LANGUAGE} code to be assessed:
{CONTENT}

Prompt for Four aspects Repository-level Annotation

You are an expert evaluator of coding. Please carefully evaluate the repository below based on the specific evaluation criteria essential for its potential use in pretraining a large language model. You are also provided with the repository's file tree to assist in assessing the overall structure and organization.

Your response must be structured strictly in **YAML format** with the following keys:

1. **Summary**: A brief summary that explains the core features, functionality, technologies used, and the overall purpose of the repository.
2. **KeyDetails**: Extract concrete implementation details that reflect the repository's architecture and design choices. Focus on how modules interact, how workflows are built across files, and which design decisions are central to the repository's functionality. Reference specific file paths, functions, or classes where possible.
 - If the code does not contain meaningful internal logic (e.g., trivial scripts or placeholder files), then set this field to **"No KeyDetails"**.
3. **Weaknesses**: Identify potential weaknesses, limitations, or areas for improvement in the repository. This may include poor modularity, lack of documentation, suboptimal performance, overly complex dependencies, or any other issues that could impact maintainability or usability.
4. **Rating**: A numerical rating of the repository on a scale from 1 to 10, where 1 indicates the lowest quality and 10 indicates the highest quality.

Do not include any explanation or commentary in the output, only the YAML with the four fields.

KeyDetails may cover (if present in the repository):

- **Cross-File Workflows**: How data, parameters, or control flow move between key modules, and explicitly explain how different modules work together to achieve the repository's functionality.
- **Architecture & Module Design**: How the repository is structured into directories and files, and how responsibilities are divided. First describe what each module's functionality is and how it is implemented, then clearly point out which modules are key and explain why they are critical.
- **Configuration & Initialization**: How configuration is handled (e.g., config files, environment variables, command-line arguments).
- **Reusability**: Shared utility functions, base classes, or components that are reused across the project.
- **Robustness & Performance**: Strategies for error handling, logging, scalability, or optimization.
- **Documentation & Testing**: Presence of README, examples, tests, or CI/CD scripts that demonstrate usage and maintainability.

Criteria for Evaluation:

* **Code Quality**:

- * Clear, readable, and maintainable code.
- * Proper commenting, meaningful naming conventions, and logical structure.
- * Consistent formatting adhering to industry standards.

* **Completeness of Functionality**:

- * Implementation covers main features expected based on repository description.
- * Functionality aligns closely with stated goals and objectives.

* **Usability**:

- * Clear and informative documentation (README, comments, wiki).
- * Easy-to-follow setup, installation, and usage instructions.
- * Availability of practical examples or demos.

* **Diversity of Tech Stack**:

- * Appropriate variety and integration of programming languages, tools, frameworks, or libraries.
- * Effective use of technologies without unnecessary complexity.

* **Project Structure Clarity**:

- * Logical and intuitive organization of files and directories.
- * Clear separation of concerns across files/modules with proper abstraction and modularity.
- * Reasonable cross-file/module dependencies with effective utilization of shared components.

After reviewing the provided repository and its file tree, generate a brief summary, extract the key implementation details (only the ones relevant in the repository or "No KeyDetails"), identify potential weaknesses, and finally conclude with the rating based on the given criteria. Your output must strictly follow the YAML format.

Example Output:

```
```yaml
Summary: [Brief summary of the repository]
KeyDetails:
 - |
 Cross-File Logic: [Description of important workflows or data flow across modules]
 - |
 Module & Architecture Design: [Explanation of directory structure, modularization, each
 module's functionality and implementation, and then highlight the key modules and why they
 are critical]
 - |
 Configuration & Setup: [Centralized config or initialization approach]
 - |
 Documentation & Testing: [Tests, examples, documentation coverage, and file tree insights]
Weaknesses:
 - [Potential weakness or area for improvement 1]
 - [Potential weakness or area for improvement 2]
Rating: [Numerical rating from 1-10]
```

REPOSITORY:
{REPOSITORY}
Repository Tree:
{REPO_TREE}
Code to be assessed:
{CONTENT}
```

Prompt for Two aspects Repository-level Annotation

You are an expert evaluator of coding. Please carefully evaluate the repository below based on the specific evaluation criteria essential for its potential use in pretraining a large language model.

Your response must be structured strictly in **YAML format** with the following keys:

1. **Summary**: Provide a **detailed summary** that comprehensively explains:
 - * The overall purpose and functionality of the repository.
 - * The main technologies, frameworks, and libraries used.
 - * The general workflow or logic of how the repository operates.
 - * The organization and structure of the codebase, including important directories or modules and their roles.
2. **Rating**: A numerical rating to the repository on a scale from 1 to 10, where 1 indicates the lowest quality and 10 indicates the highest quality.

Do not include any explanation or commentary in the output, only the YAML with the two fields.

Criteria for Evaluation:

- * **Code Quality**:
 - * Clear, readable, and maintainable code.
 - * Proper commenting, meaningful naming conventions, and logical structure.
 - * Consistent formatting adhering to industry standards.
- * **Completeness of Functionality**:
 - * Implementation covers main features expected based on repository description.
 - * Functionality aligns closely with stated goals and objectives.
- * **Usability**:
 - * Clear and informative documentation (README, comments, wiki).
 - * Easy-to-follow setup, installation, and usage instructions.
 - * Availability of practical examples or demos.
- * **Diversity of Tech Stack**:
 - * Appropriate variety and integration of programming languages, tools, frameworks, or libraries.
 - * Effective use of technologies without unnecessary complexity.
- * **Project Structure Clarity**:
 - * Logical and intuitive organization of files and directories.
 - * Clear separation of concerns across files/modules with proper abstraction and modularity.
 - * Reasonable cross-file/module dependencies with effective utilization of shared components.

After reviewing the provided repository, generate a **detailed summary** as described above. Then, based on the evaluation criteria, conclude with the rating. Your output must strictly follow the YAML format.

Example Output:

```
```yaml
Summary: [Detailed summary of the repository's purpose, technologies, workflow, and structure]
Rating: 5
```

REPOSITORY:
{REPOSITORY}
Code to be assessed:
{CONTENT}
```

Demonstration of repository details

Summary: The repository is a Flask-based web application for predicting California house prices using a trained Random Forest Regressor model. It includes a web interface for user input, an API endpoint for programmatic predictions, and a model training pipeline with hyperparameter tuning. The project is structured into modules for the web app, model

training, and templates for the UI.

KeyDetails:

- |
Cross-File Workflows: The ``applications/classification.py`` script handles data loading, preprocessing, model training, and saves the trained model as ``model.pkl``. The ``app.py`` Flask application loads this model and uses it for predictions via the ``/api/data`` endpoint. The ``templates/index.html`` template provides a UI for user input, and JavaScript in the template submits the data to the API endpoint.
- |
Module & Architecture Design: The project is divided into key modules:
 - ``app.py``: Main Flask application with routes for the web interface and API.
 - ``applications/classification.py``: Contains the model training pipeline with preprocessing, hyperparameter tuning, and model saving.
 - ``templates/index.html``: HTML template for the user interface.
 - ``run.sh``: Script for setting up the virtual environment, installing dependencies, and running the app.These modules are logically separated, with ``app.py`` and ``applications/classification.py`` being the most critical for core functionality.
- |
Configuration & Initialization: The ``requirements.txt`` file lists dependencies, and the ``run.sh`` script automates setup. The virtual environment is stored in the ``venv`` directory, and the model is saved as ``model.pkl``.
- |
Documentation & Testing: The ``README.md`` provides clear setup instructions, usage details, and API documentation. However, there are no tests or examples for the model or web application beyond the provided interface.

Weaknesses:

- The JavaScript ``modelPredict()`` function in ``templates/index.html`` is incomplete and does not fully handle the prediction workflow.
- There is no error handling or validation for the input data in the Flask API endpoint.
- No tests are provided for the model or web application to ensure correctness and robustness.
- The ``requirements.txt`` includes ``jsonify``, which is not a standalone package and should be removed.

Rating: 6

B Different Formats

LSP format

```
<|ignore_start|> #<--symbols-->
- function add(a: int, b: int) -> int
<|ignore_end|>
<|file_sep|> src/utils/add.py
from div import div
from mul import mul
def add(a, b):
    """
    This function performs addition.
    """
    return a + b

a = 1
b = 2
<|keep_start|> #<--hovers-->
def div(a, b):
    """
    This function performs division.
    """

<|keep_end|>
c = div(a,b) #<--function call-->

<|keep_start|> #<--diagnostics-->
warning: mul imported but not used
<|keep_end|>
```

FIM format

```
<|file_sep|> src/model/train.py

<|fim_prefix|>
def train(model, data):
    model.train()
    for batch in data:
<|fim_middle|>
        loss = model(batch)
        loss.backward()
<|fim_suffix|>
    optimizer.step()
    return model
```

Normal format

```
<|file_sep|> scripts/run.sh

#!/bin/bash
export CUDA_VISIBLE_DEVICES=0
python train.py --config configs/base.yaml
```

Normal format (ignored)

```
<|ignore_start|>
This file was ignored.
<|ignore_end|>
<|file_sep|> data/raw/large_dump.txt
```

Repository-level format

<|repo_name|> example-repo

<|ignore_start|>
Directory Tree:

```
.
├── src/
│   ├── utils/
│   │   ├── add.py
│   │   ├── mul.py
│   │   └── div.py
│   └── model/
│       └── train.py
├── scripts/
└── run.sh
```

Repo Summary:

This repository is a small, modular Python codebase demonstrating a simple arithmetic utility library and a minimal model training workflow. It separates low-level mathematical operations from higher-level model logic and execution scripts, serving as a lightweight example of code organization rather than a full-featured machine learning system.

KeyDetails:

- ArchitectureAndModuleDesign: >
The repository is organized into src /, which contains the core Python modules, and scripts/, which contains execution helpers. The src/utils/ directory encapsulates basic arithmetic operations, while src/model/ contains training-related logic.

Weaknesses:

- The functionality is highly trivial and does not represent a realistic or complex training pipeline suitable for large-scale experimentation.
- There is no documentation (e.g., README), inline comments, or usage examples.
- No tests, logging, or error-handling strategies are present beyond what may exist implicitly in Python.

Rating: 3

<|ignore_end|>

<|file_sep|> src/utils/math.py
(definitions and LSP-enriched content omitted)

<|file_sep|> src/model/train.py
(FIM-formatted content omitted)

<|file_sep|> scripts/run.sh
#!/bin/bash
python train.py

C Training model details

| Component | Specification |
|-------------------------|----------------------------------|
| # Layers | 16 |
| Hidden Size | 2048 |
| Intermediate Size | 8192 |
| Attention Heads | 32 |
| KV Heads | 8 |
| Head Dimension | 64 |
| Activation | SiLU |
| Normalization | RMSNorm ($\epsilon = 10^{-5}$) |
| MLP Bias | False |
| Attention Bias | False |
| Attention Dropout | 0.0 |
| Positional Encoding | RoPE |
| RoPE θ | 500,000 |
| RoPE Scaling Factor | 32.0 |
| Original Context Length | 8,192 |
| Max Context Length | 131,072 |
| Vocabulary Size | 128,256 |
| BOS / EOS Token ID | 128000 / 128001 |
| Embedding Tying | Enabled |
| Precision | bfloat16 |

Table 5: Model architecture and configuration details.

D Language distributions

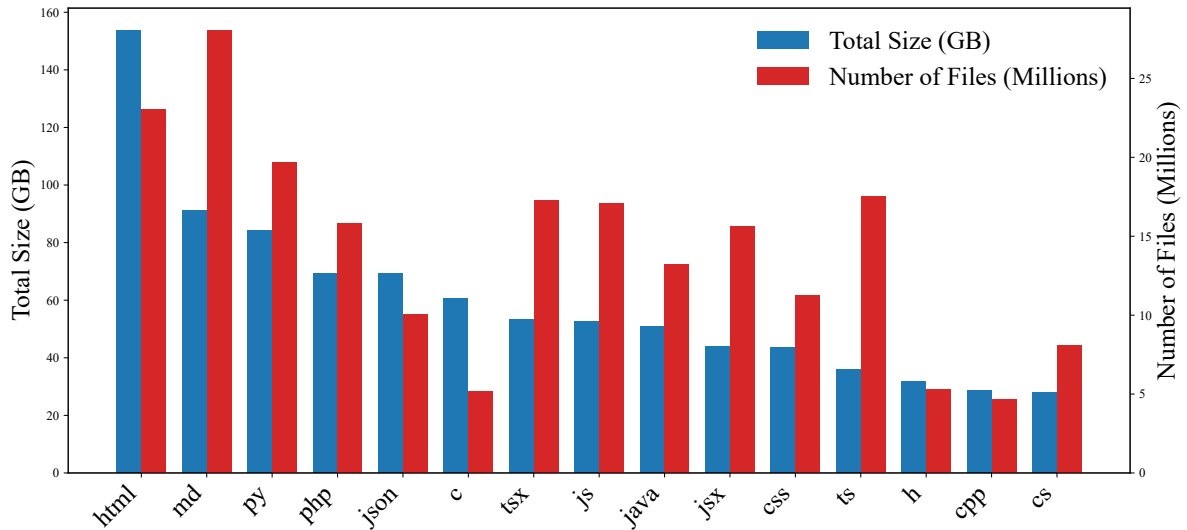


Figure 4: The distribution of top program languages in MegaCode (before data sampling).

E Evaluation results

| Model | EvalPlus | | | | MultiPL-E | | | | | | | CRUX-Eval | | AVG |
|--------------------------------------|----------|-------|-------|-------|-----------|-------|-------|-------|-------|-------|-------|-----------|--------|-------|
| | HE | HE+ | MBPP | MBPP+ | Java | C++ | JS | C# | PHP | Shell | TS | Input | Output | |
| Llama-3.2-1B | 18.30 | 15.90 | 39.10 | 32.60 | 13.92 | 15.53 | 15.53 | 17.09 | 9.94 | 1.90 | 18.24 | 29.00 | 24.20 | 19.33 |
| MegaCode w/ LSP+Summary | | | | | | | | | | | | | | |
| 1B/10B | 19.50 | 16.50 | 39.30 | 31.60 | 13.29 | 13.04 | 17.39 | 19.62 | 9.32 | 1.90 | 16.35 | 28.10 | 23.10 | 19.15 |
| 2B/10B | 21.30 | 17.70 | 39.10 | 31.60 | 14.56 | 14.91 | 17.39 | 17.09 | 9.94 | 2.53 | 15.72 | 30.50 | 21.50 | 19.53 |
| 3B/10B | 19.50 | 17.10 | 42.10 | 33.30 | 14.56 | 14.29 | 16.15 | 18.35 | 9.94 | 3.16 | 14.47 | 30.00 | 23.80 | 19.75 |
| 4B/10B | 20.10 | 18.30 | 39.30 | 32.10 | 12.66 | 13.66 | 16.15 | 22.15 | 11.18 | 3.16 | 18.24 | 30.90 | 22.40 | 20.02 |
| 5B/10B | 18.90 | 16.50 | 42.90 | 33.60 | 15.19 | 14.91 | 18.63 | 15.19 | 9.94 | 1.27 | 16.35 | 30.10 | 23.50 | 19.77 |
| 6B/10B | 20.70 | 17.10 | 40.10 | 32.80 | 14.56 | 16.77 | 18.63 | 16.46 | 11.18 | 1.90 | 18.87 | 31.00 | 22.80 | 20.22 |
| 7B/10B | 20.10 | 14.60 | 40.60 | 32.10 | 13.29 | 15.53 | 18.01 | 16.46 | 11.80 | 1.90 | 18.24 | 30.90 | 22.80 | 19.72 |
| 8B/10B | 19.50 | 15.90 | 40.90 | 33.30 | 13.92 | 16.77 | 16.77 | 16.46 | 12.42 | 3.16 | 20.75 | 31.10 | 23.60 | 20.35 |
| 9B/10B | 19.50 | 15.20 | 42.60 | 34.10 | 13.29 | 16.77 | 18.63 | 18.35 | 11.18 | 2.53 | 19.50 | 32.00 | 23.40 | 20.54 |
| 10B/10B | 22.00 | 17.70 | 41.10 | 32.60 | 15.19 | 17.39 | 18.63 | 16.46 | 10.56 | 2.53 | 20.75 | 31.90 | 23.20 | 20.77 |
| MegaCode w/ LSP | | | | | | | | | | | | | | |
| 1B/10B | 19.50 | 16.50 | 40.40 | 31.30 | 13.29 | 13.66 | 16.77 | 18.35 | 11.18 | 2.53 | 16.35 | 27.30 | 23.90 | 19.31 |
| 2B/10B | 18.90 | 15.90 | 40.10 | 32.30 | 13.29 | 12.42 | 16.77 | 18.35 | 8.07 | 2.53 | 15.72 | 30.00 | 22.00 | 18.95 |
| 3B/10B | 18.90 | 15.90 | 38.80 | 31.60 | 13.29 | 16.15 | 14.91 | 16.46 | 9.94 | 1.90 | 17.61 | 30.10 | 22.50 | 19.08 |
| 4B/10B | 22.60 | 17.70 | 40.90 | 32.60 | 13.29 | 16.77 | 16.77 | 17.09 | 10.56 | 1.90 | 17.61 | 28.70 | 23.10 | 19.97 |
| 5B/10B | 21.30 | 17.10 | 39.80 | 32.10 | 13.29 | 19.25 | 18.01 | 16.46 | 12.42 | 1.90 | 17.61 | 28.40 | 21.90 | 19.96 |
| 6B/10B | 17.10 | 12.80 | 40.10 | 32.60 | 13.29 | 16.77 | 18.63 | 17.09 | 11.80 | 1.90 | 16.98 | 28.50 | 22.80 | 19.26 |
| 7B/10B | 19.50 | 17.10 | 39.80 | 32.80 | 12.66 | 17.39 | 18.01 | 16.46 | 11.18 | 3.16 | 16.98 | 30.20 | 24.10 | 19.95 |
| 8B/10B | 18.90 | 15.90 | 40.40 | 32.80 | 15.82 | 16.77 | 18.63 | 16.46 | 9.94 | 2.53 | 18.87 | 30.60 | 23.50 | 20.09 |
| 9B/10B | 20.10 | 16.50 | 39.60 | 32.30 | 14.56 | 14.29 | 16.77 | 17.72 | 9.94 | 2.53 | 16.98 | 30.50 | 24.20 | 19.69 |
| 10B/10B | 20.70 | 16.50 | 41.10 | 33.30 | 14.56 | 15.53 | 16.77 | 18.99 | 10.56 | 2.53 | 17.61 | 29.90 | 23.60 | 20.13 |
| MegaCode w/ Summary | | | | | | | | | | | | | | |
| 1B/10B | 17.10 | 14.60 | 38.60 | 29.60 | 11.39 | 9.94 | 17.39 | 19.62 | 9.94 | 1.90 | 16.35 | 26.80 | 23.10 | 18.18 |
| 2B/10B | 18.30 | 15.20 | 40.60 | 31.10 | 13.29 | 16.15 | 14.91 | 17.09 | 10.56 | 1.90 | 16.98 | 30.60 | 23.20 | 19.22 |
| 3B/10B | 20.70 | 16.50 | 38.60 | 30.30 | 13.29 | 14.29 | 16.15 | 12.03 | 11.80 | 1.27 | 16.98 | 31.40 | 23.50 | 18.99 |
| 4B/10B | 20.70 | 17.70 | 39.10 | 30.80 | 15.82 | 14.91 | 18.01 | 17.72 | 10.56 | 3.16 | 18.87 | 32.20 | 23.50 | 20.23 |
| 5B/10B | 22.60 | 18.30 | 38.80 | 31.80 | 14.56 | 14.91 | 19.25 | 19.62 | 11.18 | 3.16 | 20.13 | 30.80 | 22.40 | 20.58 |
| 6B/10B | 19.50 | 15.20 | 37.80 | 30.10 | 15.19 | 15.53 | 14.91 | 15.82 | 11.80 | 1.90 | 17.61 | 30.60 | 22.90 | 19.14 |
| 7B/10B | 22.60 | 15.90 | 41.90 | 32.80 | 15.19 | 16.15 | 14.91 | 15.82 | 13.04 | 2.53 | 17.61 | 30.80 | 22.40 | 20.13 |
| 8B/10B | 21.30 | 17.70 | 40.10 | 30.80 | 12.66 | 15.53 | 17.39 | 16.46 | 12.42 | 2.53 | 18.87 | 30.10 | 21.90 | 19.83 |
| 9B/10B | 22.60 | 18.30 | 41.10 | 32.30 | 12.66 | 16.15 | 17.39 | 17.72 | 13.66 | 3.16 | 19.50 | 30.10 | 22.00 | 20.51 |
| 10B/10B | 23.80 | 17.10 | 39.80 | 31.80 | 15.19 | 15.53 | 16.77 | 18.99 | 12.42 | 2.53 | 18.87 | 29.80 | 22.00 | 20.35 |
| MegaCode w/ LSP+Summary (in the end) | | | | | | | | | | | | | | |
| 1B/10B | 19.50 | 17.10 | 39.60 | 30.60 | 12.03 | 13.04 | 15.53 | 20.89 | 11.80 | 1.27 | 16.35 | 27.60 | 23.40 | 19.13 |
| 2B/10B | 17.10 | 15.90 | 39.10 | 30.10 | 11.39 | 12.42 | 16.15 | 16.46 | 8.07 | 4.43 | 15.72 | 30.10 | 23.10 | 18.46 |
| 3B/10B | 19.50 | 17.10 | 39.30 | 31.10 | 12.66 | 15.53 | 17.39 | 17.09 | 8.70 | 3.16 | 16.98 | 31.50 | 23.50 | 19.50 |
| 4B/10B | 20.10 | 17.10 | 41.40 | 32.10 | 14.56 | 14.29 | 18.63 | 16.46 | 10.56 | 2.53 | 18.87 | 30.80 | 21.50 | 19.92 |
| 5B/10B | 22.60 | 18.90 | 39.80 | 32.30 | 14.56 | 14.29 | 17.39 | 15.19 | 13.04 | 3.16 | 18.87 | 31.80 | 22.00 | 20.30 |
| 6B/10B | 20.70 | 17.10 | 40.60 | 32.60 | 15.82 | 14.91 | 17.39 | 15.19 | 13.66 | 1.90 | 18.24 | 31.90 | 23.40 | 20.26 |
| 7B/10B | 20.10 | 17.10 | 40.60 | 31.60 | 15.19 | 16.77 | 16.77 | 15.19 | 13.04 | 3.16 | 16.35 | 32.50 | 22.80 | 20.09 |
| 8B/10B | 19.50 | 15.90 | 40.10 | 32.80 | 13.92 | 18.63 | 16.77 | 19.62 | 13.04 | 3.80 | 18.24 | 31.50 | 22.90 | 20.52 |
| 9B/10B | 18.90 | 15.90 | 41.60 | 33.30 | 15.19 | 19.25 | 17.39 | 19.62 | 14.29 | 2.53 | 18.24 | 31.80 | 22.90 | 20.84 |
| 10B/10B | 20.70 | 17.10 | 40.10 | 33.60 | 15.19 | 18.63 | 17.39 | 15.82 | 13.04 | 1.90 | 16.98 | 32.20 | 22.90 | 20.43 |
| SwallowCode | | | | | | | | | | | | | | |
| 1B/10B | 21.30 | 18.30 | 34.80 | 28.80 | 13.29 | 11.80 | 12.42 | 13.92 | 7.45 | 1.27 | 13.21 | 32.90 | 24.50 | 18.00 |
| 2B/10B | 20.70 | 18.30 | 33.80 | 28.30 | 10.76 | 10.56 | 14.29 | 18.35 | 9.94 | 2.53 | 14.47 | 32.00 | 23.60 | 18.28 |
| 3B/10B | 20.70 | 17.70 | 37.10 | 30.80 | 10.76 | 11.80 | 15.53 | 20.25 | 8.70 | 1.27 | 14.47 | 32.60 | 22.60 | 18.79 |
| 4B/10B | 22.60 | 19.50 | 39.60 | 32.60 | 12.66 | 13.04 | 14.91 | 16.46 | 10.56 | 2.53 | 13.21 | 31.40 | 22.40 | 19.34 |
| 5B/10B | 24.40 | 21.30 | 37.30 | 31.80 | 11.39 | 13.66 | 13.66 | 18.99 | 9.32 | 2.53 | 16.35 | 31.50 | 22.20 | 19.57 |
| 6B/10B | 21.30 | 18.90 | 39.10 | 32.30 | 10.13 | 14.29 | 15.53 | 18.99 | 11.18 | 2.53 | 16.98 | 29.10 | 21.90 | 19.40 |
| 7B/10B | 23.20 | 19.50 | 41.10 | 35.10 | 10.13 | 13.66 | 13.66 | 19.62 | 8.07 | 1.27 | 16.98 | 29.80 | 21.20 | 19.48 |
| 8B/10B | 23.80 | 20.10 | 42.90 | 35.60 | 10.76 | 13.66 | 12.42 | 18.99 | 11.80 | 1.27 | 16.98 | 29.90 | 21.50 | 19.98 |
| 9B/10B | 24.40 | 20.10 | 42.10 | 36.10 | 10.13 | 12.42 | 14.29 | 20.25 | 9.32 | 1.27 | 16.35 | 28.70 | 22.10 | 19.81 |
| 10B/10B | 22.60 | 19.50 | 41.10 | 34.80 | 10.13 | 12.42 | 14.29 | 18.35 | 11.80 | 1.90 | 16.35 | 29.60 | 21.60 | 19.57 |

Table 6: Detailed results on code generation benchmarks for different continued pretraining settings. We report per-benchmark scores on EvalPlus, MultiPL-E (seven languages), CRUXEval, and overall average (AVG).