Not All Code Is Equal: A Data-Centric Study of Code Complexity and LLM Reasoning

Anonymous authors
Paper under double-blind review

Abstract

Large Language Models (LLMs) increasingly exhibit strong reasoning abilities, often attributed to their capacity to generate chain-of-thought-style intermediate reasoning. Recent work suggests that exposure to code can further enhance these skills, but existing studies largely treat code as a generic training signal, leaving open the question of which properties of code actually contribute to improved reasoning. To address this gap, we study the *structural complexity* of code, which captures control flow and compositional structure that may shape how models internalise multi-step reasoning during fine-tuning. We examine two complementary settings: *solution-driven complexity*, where structural complexity varies across multiple solutions to the same problem, and *problem-driven complexity*, where structural complexity reflects variation in the underlying tasks. Using cyclomatic complexity and logical lines of code to construct controlled fine-tuning datasets, we evaluate a range of open-weight LLMs on diverse reasoning benchmarks. Our findings show that although code can improve reasoning, its usefulness is substantially shaped by structural properties. In 83% of experiments, restricting fine-tuning data to a specific structural complexity range outperforms training on structurally diverse code, pointing to a *data-centric path for improving reasoning beyond scaling*.

1 Introduction

Large language models (LLMs) have rapidly evolved from surface-level language processing systems into capable problem solvers, exhibiting increasingly strong reasoning behaviours across mathematical, logical, and multi-disciplinary tasks (Wei et al., 2022a; Huang & Chang, 2023). A large body of work attributes these gains to the use of Chain-of-Thought (CoT) style explanations, in which models generate intermediate reasoning steps before producing an answer (Wei et al., 2022b; Wang et al., 2023). CoTs have been extensively studied in natural language settings, with recent theoretical analyses providing insight into why structured intermediate traces benefit model reasoning (Feng et al., 2023), suggesting that symbolic scaffolding reduces search complexity and stabilises problem decomposition.

A related "program-of-thought" phenomenon has been observed for code, where programs provide explicit reasoning structure that would otherwise be expressed in natural language (Chen et al., 2023). This is because code naturally expresses control flow, branching, and intermediate computation–structures that are useful for multi-step reasoning–and has consequently been used as a structured signal to encourage CoT (Lin et al., 2025). Beyond inference-time scaffolding, exposure to code during training has also been shown to enhance reasoning more broadly: models trained on code data often demonstrate improved multi-step reasoning and quantitative problem solving compared to models trained solely on natural language (Zhang et al., 2024b; Waheed et al., 2026; Yang et al., 2025).

Despite this emerging evidence, the properties of code that contribute to these gains remain under-explored. Existing studies largely treat code as an undifferentiated training signal, without examining which characteristics of the code are actually important (Aryabumi et al., 2024; Zhang et al., 2024b; Waheed et al., 2026). To fill this gap, we ask whether *fine-grained, measurable properties* of code can systematically shape the downstream reasoning performance of LLMs when used for fine-tuning. We focus on *structural complexity*

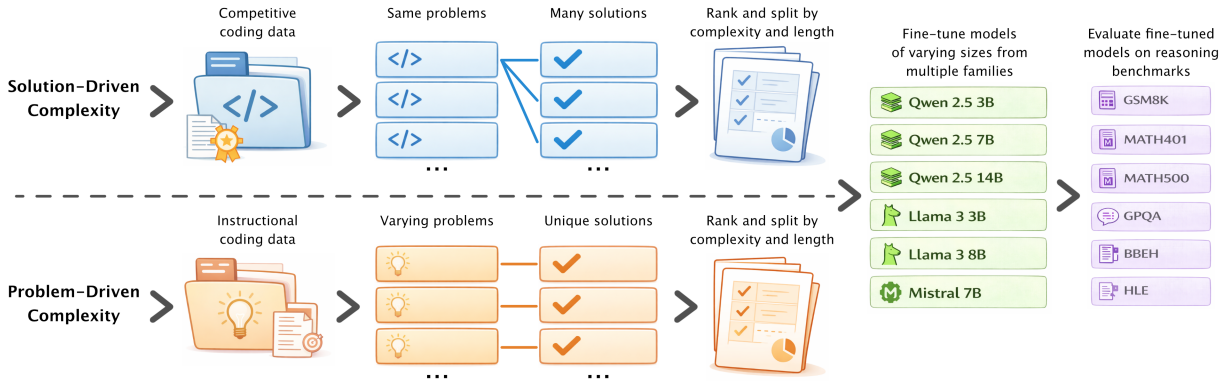

Figure 1: ***Overview of our experimental pipeline.*** We construct two complementary datasets that control the structural complexity of code through *solution-driven* (top) and *problem-driven* (bottom) settings, creating twelve splits for each (five different levels and one control, for two different structural complexity metrics). We then use these splits to fine-tune six models from three model families and evaluate downstream reasoning performance across six widely-used reasoning benchmarks.

as a candidate driver of code-induced reasoning gains, motivated by the fact that more structurally complex programs exhibit deeper branching and richer execution paths (Fenton & Bieman, 2014), which can be viewed as an implicit form of a structured reasoning trace that exposes models to multi-step decomposition patterns during fine-tuning.

Specifically, we investigate this question through two complementary settings (Figure 1): ***(i)*** *Solution-driven complexity*, where complexity arises from the structure of the code itself. We use multiple solutions to identical programming problems to vary complexity while holding tasks fixed. ***(ii)*** *Problem-driven complexity*, this setting reflects variation in the underlying tasks themselves, where different prompts are naturally paired with reference solutions of differing structural complexity. Together, these settings allow us to disentangle the effects of code complexity arising from how code is written versus which problem the code is solving. We construct two datasets with controlled complexity variation in `Python`, `JavaScript` and `Java`, using cyclomatic complexity (McCabe, 1976) and logical lines of code (Nguyen et al., 2007) as complementary structural metrics. We then fine-tune a diverse set of open-weight models across multiple parameter scales and evaluate their reasoning performance on publicly available benchmarks spanning mathematical and multi-disciplinary problem solving.

Our results reveal that *not all code is equal*. ***(1)*** Code fine-tuning does not yield uniform reasoning gains: even when using code datasets previously shown to be effective, improvements vary substantially across models and benchmarks, and depend strongly on the structural complexity of the fine-tuning data. ***(2)*** The relationship between code complexity and reasoning performance is strongly non-monotonic and model-dependent, with intermediate complexity ranges often performing best. ***(3)*** Control datasets that mix code across all complexity levels are rarely optimal: in 83% of experiments, restricting fine-tuning to a specific complexity range yields better reasoning performance than training on a diverse code corpus.

Our results indicate that the usefulness of code as a training signal is sensitive to its structural properties. This challenges the prevailing assumption that greater diversity or quantity of code is inherently beneficial (Liu et al., 2025; Abed et al., 2025), and instead points to a data-centric alternative: carefully selecting or constructing code with appropriate structural complexity matched to the specific model. Because high-quality code data is expensive to collect and train on (Chen et al., 2025), understanding which code structures are associated with stronger reasoning offers a practical path to improving LLM reasoning beyond simply scaling models or datasets.

Our contributions are as follows:

- We present the first systematic study of how *structural properties of code* – specifically cyclomatic complexity and logical lines of code – affect the reasoning abilities of LLMs during fine-tuning.
- We provide empirical evidence that reasoning gains from code are *non-uniform* and *non-monotonic*, and that restricting fine-tuning data to model-specific complexity ranges often outperforms training on structurally diverse code.
- We publicly release our complexity-controlled datasets[1] to support reproducibility and encourage further research on the interaction between code complexity and LLM reasoning.

2 Related Work

Reasoning abilities of LLMs. As LLMs evolve, reasoning and problem solving have emerged as core abilities, often appearing once models reach sufficient scale (Wei et al., 2022a). Early work showed that sufficiently large models can exhibit multi-step reasoning when prompted to generate intermediate *chain-of-thought* (Wei et al., 2022b), an observation that catalysed substantial research into understanding and improving reasoning performance (Feng et al., 2023; Huang & Chang, 2023). Reasoning can be improved through a range of approaches, including prompt engineering (Kojima et al., 2022), novel decoding strategies (Wang et al., 2023), fine-tuning (Trung et al., 2024), and reinforcement learning (Wang et al., 2025). Evaluation practices have evolved alongside these methods. Multi-step reasoning benchmarks such as GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021) are widely used to assess arithmetic and logical reasoning, while more recent benchmark suites, including BIG-BENCH EXTRA HARD (Kazemi et al., 2025) and ARC-AGI (Chollet et al., 2025), probe a broader range of complex and abstract reasoning behaviours.

Using code to enhance LLM reasoning. Using code and code-instruction data has not only been shown to consistently improve LLM performance on code-generation and related tasks (Rozière et al., 2024; Wei et al., 2024), but also to improve general reasoning capabilities (Yang et al., 2025). Recent studies investigate this phenomenon from complementary perspectives. Some examine how the inclusion or perturbation of code data during fine-tuning affects reasoning performance (Zhang et al., 2024b; Waheed et al., 2026), while others analyse the role of code earlier in the training pipeline, such as during pre-training or continued pre-training (Ma et al., 2023; Aryabumi et al., 2024). A related line of work explores explicitly converting reasoning traces into code-like representations to leverage code as a structured training signal (Lin et al., 2025). An alternative line of work explores *program-of-thought* prompting, where intermediate reasoning is externalised into executable code to reduce cognitive load and improve error handling in symbolic tasks (Chen et al., 2023). Overall, prior work demonstrates that code can improve reasoning, but offers limited insight into which properties of code best drive these gains–an omission our work addresses by systematically varying code complexity during fine-tuning.

Code complexity metrics. The analysis of program structure via static code metrics has a long history in software engineering (Fenton & Bieman, 2014). Foundational measures such as cyclomatic complexity (McCabe, 1976) and Halstead metrics (Halstead, 1977) quantify structural and control-flow properties of programs, while a broader ecosystem of metrics captures complementary aspects including program size (Nguyen et al., 2007), maintainability (Coleman et al., 1994), and coupling or cohesion (Tiwari & Rathore, 2018). These classical metrics continue to play a role in modern machine learning for code. Recent work on complexity-aware code generation uses suites of established metrics to analyse and guide LLM behaviour, showing that explicit complexity signals can improve performance (Sepidband et al., 2025). Complexity metrics have also been used as discriminative features for vulnerability assessment (Tehrani & Hashemi, 2025) and automatic defect detection (Cernau et al., 2025), reinforcing their utility. Beyond these targeted applications, recent studies have evaluated LLMs' ability to reason about code maintainability under controlled complexity conditions (Dillmann et al., 2024). Collectively, these lines of work support viewing code complexity metrics as interpretable, quantitative signals–motivating our study of how varying complexity in fine-tuning data affects reasoning behaviour in LLMs.

[1] *redacted for review*

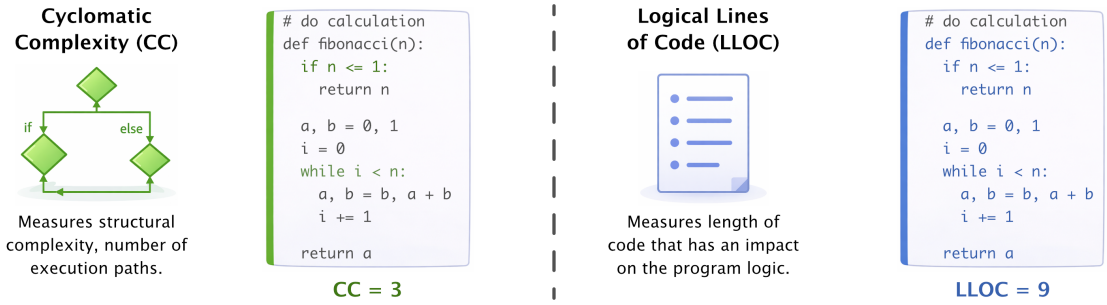

Figure 2: ***Comparison of the code metrics used in this study.*** We calculate both metrics for our *solution-driven complexity* dataset (CODENET) and our *problem-driven complexity* dataset (INSTRUCT). Together they allow us to disentangle the effects of structural complexity from size-based complexity.

3 Methodology

3.1 Problem Formulation

This work investigates whether the *structural complexity* of code used during fine-tuning influences the reasoning abilities of LLMs. Prior studies have shown that exposure to code can improve reasoning (Zhang et al., 2024b; Waheed et al., 2026), yet the properties of the code that drive these gains remain poorly understood. We study this question through two complementary notions of structural complexity, which we define for the purposes of this work. In *solution-driven complexity*, structural complexity arises from the structure of the code itself: we use multiple solutions to the same underlying problems to isolate code complexity as a variable. In contrast, problem-driven complexity does not hold task semantics fixed. Instead, we stratify naturally occurring instruction–solution pairs by the structural complexity of their reference code. This setting captures cases where code structure varies together with the underlying task distribution, rather than across alternative implementations of the same task. By disentangling these two sources of complexity and controlling them through dataset construction, we aim to isolate how fine-tuning on code of varying structural properties affects downstream reasoning performance.

3.2 Code Complexity Metrics

Metric selection. Code complexity has long been studied in software engineering through static analysis, where quantitative metrics are used to characterise structural properties of programs such as control flow and size (Fenton & Bieman, 2014). In this work, we are interested in complexity measures that plausibly relate to reasoning: intuitively, code with richer control flow and more elaborate structure may expose models to patterns that resemble multi-step reasoning. Guided by this motivation, and because this is an initial controlled study of code-complexity effects on reasoning, we deliberately focus on simple, interpretable, and widely established metrics (Figure 2). First, we use *cyclomatic complexity* (CC), which captures the number of independent execution paths through a program (McCabe, 1976). Although CC has known limitations, it remains one of the most widely used and enduring software complexity metrics (Siahaan et al., 2025), making it a natural starting point for studying whether control-flow structure affects reasoning-oriented fine-tuning. Second, we use *logical lines of code* (LLOC), a simpler size-based metric that reflects the amount of executable logic independent of formatting or comments (Nguyen et al., 2007). Because CC tends to increase with code length, incorporating LLOC enables us to compare control-flow complexity with a complementary size-based signal.

Metric calculation. We construct multilingual training datasets to capture diverse programming problems and solution patterns, focussing on the three most used programming languages on GitHub: `Python`, `JavaScript` (including `TypeScript`), and `Java` (GitHub Staff, 2025). To ensure consistency and reproducibility across languages, we rely on established open-source static analysis tools for metric computation. For `Python`, we use `Radon` (rubik, 2025); for `JavaScript`, we use `escomplex` (escomplex, 2025); and for

`Java`, we use `PMD` (pmd, 2025). All tools are applied using a uniform preprocessing pipeline. *Implementation details are provided in Appendix A.*

3.3 Dataset Construction

Solution-driven complexity. We require a dataset containing many distinct code solutions to the same underlying problems, allowing complexity to vary independently of problem semantics. We therefore adopt `Project CodeNet` (Puri et al., 2021), a large-scale corpus of competitive programming problems paired with thousands of accepted solutions across multiple programming languages. CodeNet is well suited to our setting, as it provides diverse, independently authored solutions to identical problem statements, enabling controlled comparisons of code complexity while holding the task fixed. CodeNet problem descriptions are provided as HTML, and solutions are presented as standalone code snippets, neither of which is directly suitable for instruction-style fine-tuning. We therefore augment the dataset, using an LLM to convert HTML problem statements into concise natural language instructions and to wrap each solution in a consistent response format. We use the `gpt-5-mini-2025-08-07` model for this augmentation step, as it is a cost-effective state-of-the-art model that performs reliably on well-defined tasks (OpenAI, 2025). *Augmentation prompts are detailed in Appendix B.*

For each problem, we compute CC and LLOC over all available `Python`, `JavaScript`, and `Java` solutions. For both metrics, we split the dataset by ranking solutions within each problem–language pair and selecting five representative solutions: the least complex, the most complex, and three evenly spaced solutions from the remainder of the distribution. These selections form five complexity-controlled dataset splits corresponding to increasing levels of solution-driven complexity (splits named: MIN, LOW, MID, HIGH, MAX). In addition, we construct a control dataset (CTRL) by sampling solutions uniformly across these complexity levels. The resulting CODENET dataset comprises twelve splits (six per metric) of 8,087 samples.

Problem-driven complexity. Next, we construct a dataset in which task semantics are not held fixed, and code complexity varies across naturally occurring instruction–solution pairs rather than across alternative solutions to the same problem. To do this, we gather three high-quality instruction–response code datasets–`Magicoder` (ise-uiuc, 2023), `Evol-Instruct` (nickrosh, 2024), and `WizardLM` (rombodawg, 2023)–which have been previously used for training LLMs (Wei et al., 2024) and improving their reasoning (Waheed et al., 2026). Unlike CodeNet, these datasets pair each natural language instruction with a single reference solution. As a result, they do not allow us to vary implementation structure while holding the task fixed; instead, they reflect a more natural instruction-tuning setting in which task semantics and reference-code structure vary together.

For each dataset response, we use regular expressions to extract code blocks and identify the programming language. We retain only samples containing `Python`, `JavaScript`, or `Java` code and compute CC and LLOC for each. After filtering, the dataset contains 77,686 `Python`, 13,949 `JavaScript` and 8,054 `Java` samples. For each metric, we rank samples independently within each language and partition them into five disjoint complexity bins. To construct balanced datasets comparable in size to CODENET, we include all available `JavaScript` and `Java` samples and supplement them with `Python` samples until each split contains 8,087 samples (splits named: MIN, LOW, MID, HIGH, MAX). In addition, we construct a control dataset (CTRL) that samples across languages and complexity levels. The resulting INSTRUCT dataset again comprises twelve splits (six per metric), enabling controlled comparisons with CODENET.

Natural language baseline. To disentangle the effects of code exposure from general fine-tuning, we include a natural language (NL) baseline, following the methodology of prior work studying the impact of code in fine-tuning (Zhang et al., 2024b). Specifically, we reuse the same non-code ShareGPT dataset employed in that study, as it is comparable in scale to our code datasets and enables a controlled comparison. From this corpus, we sample 8,087 records to match the exact size of each code-based split. The resulting dataset therefore isolates the effect of fine-tuning itself, which is important because fine-tuning alone has been shown to sometimes induce non-trivial shifts in downstream reasoning behaviour (Luo et al., 2025). This allows us to attribute any observed reasoning changes specifically to properties of the code rather than to fine-tuning alone.

Full dataset statistics are available in Appendix C, the final datasets are publicly available on Hugging Face[2].

3.4 Evaluation Set-up

Model selection. We evaluate our approach across a diverse set of models to assess the impact of code complexity more broadly. Specifically, we select models spanning multiple sizes from three major families: `Qwen-2.5` `3B`, `7B` and `14B` (Qwen et al., 2025); `Llama-3` `3B` and `8B` (Grattafiori et al., 2024); and `Mistral-7B` (Jiang et al., 2023). These families were chosen due to their open availability, their widespread use in academic evaluation, and their coverage of similar parameter scales (`3B`–`14B`), allowing us to study whether complexity-related effects generalise across model sizes and architectures.

Training configurations. All models are fine-tuned for two epochs over each dataset split using LoRA (Hu et al., 2021) with a learning rate of 2×10^{-5}, AdamW optimisation, and a cosine learning rate schedule with a warm-up ratio of 0.1. We adopt LoRA as a standard parameter-efficient fine-tuning method, which has been shown to perform well in low-data settings and preserve base model capabilities during adaptation (Biderman et al., 2024; Zhang et al., 2024a). This makes it well-suited for our goal of isolating *data-centric effects*, as it reduces confounding factors such as catastrophic forgetting while maintaining a controlled training setup. Training is performed on NVIDIA A100 GPUs, and all experiments use the same optimiser, scheduling, and adaptation configuration to ensure comparability.

Evaluation benchmarks. To measure downstream reasoning performance, we evaluate on six publicly available benchmarks that cover a broad spectrum of reasoning demands. We first consider math-focused benchmarks that emphasise multi-step computation and logical reasoning: GSM8K, a widely used dataset of grade-school arithmetic problems (Cobbe et al., 2021); MATH401, which contains 401 arithmetic reasoning problems (Yuan et al., 2023); and MATH500, comprising 500 mathematical problems covering a broader range of topics and difficulty levels (Lightman et al., 2023). Beyond mathematics, GPQA evaluates graduate-level quantitative reasoning, including physics-based problem solving (Rein et al., 2023); BBEH-MINI is a curated subset of BIG-BENCH EXTRA HARD designed to probe complex and diverse reasoning behaviours (Kazemi et al., 2025); and HLE (Humanity's Last Exam) spans multi-disciplinary questions across the humanities, sciences, and general knowledge domains (Phan et al., 2025).

Further evaluation details are available in Appendix D.

4 Results

We begin by presenting high-level findings that are robust across models, datasets, and benchmarks. We then turn to a more fine-grained analysis that unpacks *why* these patterns arise, and how they differ across model families, dataset constructions, and reasoning tasks. Because individual model–benchmark differences can be small, our claims focus on recurring patterns across models, datasets, metrics, and benchmarks, rather than on any single pairwise comparison. *Appendix E reports complete results for all models and benchmark datasets, with bootstrap confidence intervals.*

4.1 Main Findings

Code fine-tuning yields non-uniform reasoning gains. Figure 3 shows average reasoning accuracy across six benchmarks after fine-tuning on code data split by cyclomatic complexity (CC) or logical lines of code (LLOC), under both the solution-driven (CODENET) and problem-driven (INSTRUCT) settings. Across models, fine-tuning on code often improves reasoning relative to the natural language (NL) baseline; however, these gains are clearly *not uniform*. The same model can benefit substantially from one code subset, yet show negligible–or even negative–changes when fine-tuned on code of a different structural complexity.

All experiments use small, tightly controlled fine-tuning datasets (8,087 samples per split) to ensure strict comparability across complexity levels. Under this regime, code–even when drawn from datasets previously

[2] *redacted for review*

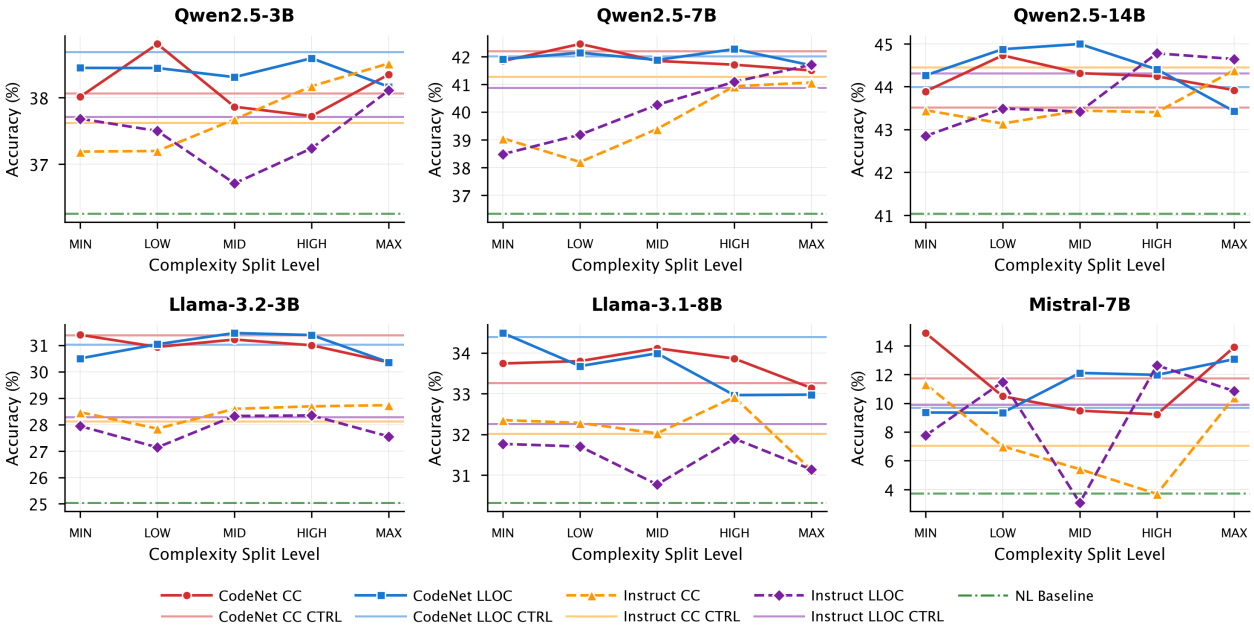

Figure 3: ***Per-model reasoning performance across complexity splits.*** Reasoning accuracy for each model after fine-tuning on complexity-controlled dataset splits. Solid lines correspond to solution-driven complexity splits (CodeNet), dashed lines to problem-driven complexity splits (Instruct); horizontal lines indicate the corresponding results for control (ctrl) datasets with mixed complexity; the dash-dotted green line denotes the model's natural language (NL) baseline after fine-tuning on a strictly non-code dataset.

shown to support reasoning (Yang et al., 2025)–is not a guaranteed source of improvement across models or benchmarks. Instead, performance exhibits pronounced sensitivity to structural complexity, with sharp peaks and troughs across adjacent splits. Taken together, these results show that the effectiveness of code fine-tuning depends critically on *which* code is used, rather than merely on the presence of code itself.

Reasoning gains are non-monotonic and model-dependent. Across all model families, the relationship between code complexity and downstream reasoning performance is strongly *non-monotonic*. Rather than improving steadily as complexity increases, accuracy curves exhibit clear peaks and troughs across both CC and LLOC splits (Figures 3 and 4). Intermediate complexity ranges often perform best, particularly for the Qwen and Llama families, but this is not universal: `Mistral-7B` exhibits a distinct pattern, benefiting more from the lowest and highest CC splits than from intermediate ones. Importantly, this pattern reflects a relative trade-off rather than a uniformly positive effect of increasing complexity. This indicates that code appears to be most useful as a training signal for reasoning when its structural complexity falls within a model-dependent effective range.

Mixed-complexity control datasets are rarely optimal. A striking and consistent finding is that control datasets–constructed by uniformly mixing code across all complexity levels–are almost never optimal. Across 20 of the 24 model–dataset combinations, at least one restricted complexity split outperforms its corresponding control (Figure 3). This finding directly challenges the common assumption that diversity in code corpora is inherently beneficial. Instead, our results suggest that *targeted restriction to a specific complexity range* often yields better reasoning performance than broad mixing. Notably, this effect holds across both solution-driven and problem-driven settings, indicating that it is not specific to a single dataset construction, but a recurring pattern across our settings.

To test whether these effects disappear with substantially more data, we additionally fine-tune on the full Instruct CC mixture, which combines all five complexity buckets (∼40k samples). Targeted complexity buckets still outperform the larger mixed dataset for both tested models (Appendix F). For `Qwen2.5-7B`, the

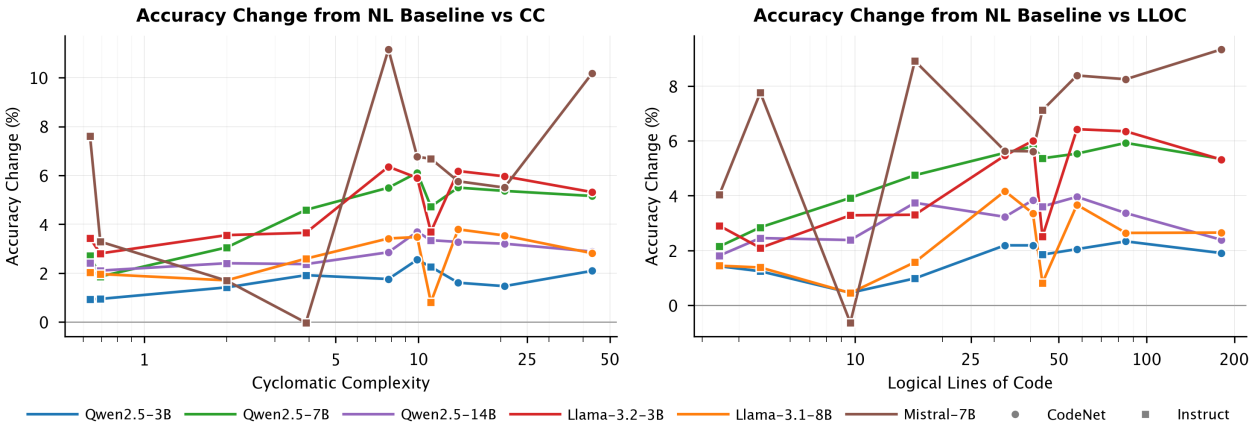

Figure 4: ***Average reasoning change vs. code complexity.*** Average accuracy change compared to the NL baseline across all six reasoning benchmarks as a function of cyclomatic complexity (CC, left) and logical lines of code (LLOC, right). Results are shown as a single line that includes both the solution-driven (CodeNet; circles) and the problem-driven (Instruct; squares) complexity datasets.

best individual bucket improves average accuracy by 4.89% over full-dataset training; for `Llama-3.2-3B`, the average gain is smaller (+0.66), but the best bucket still outperforms the full dataset on every benchmark. This suggests that the advantage of targeted complexity selection is not merely an artefact of small training splits: in these settings, adding more mixed-complexity code does not recover the performance of the best restricted subset.

Summary of main findings.

Overall, these results identify structural code complexity as a critical–and previously under-explored–factor in code-based fine-tuning for reasoning. The central takeaway is simple but consequential: fine-tuning on code does not guarantee reasoning gains, and restricting training data to an appropriate complexity or length range can yield stronger improvements than training on a large, mixed code corpus.

4.2 Fine-Grained Analysis Across Models and Datasets

Task-varying structural complexity shows stronger and more consistent effects. Figure 5 reports Spearman correlations between training-data complexity and downstream benchmark accuracy. Across both CC and LLOC, Instruct exhibits more consistent positive correlations than CodeNet. This suggests that exposure to problems that *require* more structurally complex solutions is more consistently associated with reasoning improvements than simply training on arbitrarily complex code. This pattern is particularly pronounced for the `Qwen` family, which shows predominantly positive correlations across multiple benchmarks under the Instruct setting. Consistent with this observation, Figure 3 shows clearer upward trends in the Instruct accuracy curves, especially for `Qwen2.5-7B`.

Cyclomatic complexity provides a more reliable and interpretable signal than logical lines of code. Although both CC and LLOC capture aspects of structural complexity, their effects on reasoning differ in stability and interpretability. Across models and datasets, performance trends with respect to CC are generally smoother and more consistent than those observed for LLOC. In contrast, LLOC-based splits often exhibit sharper fluctuations and less regular behaviour, particularly at higher complexity levels. This difference likely reflects the nature of the metrics themselves. CC directly captures branching structure and control flow, which are closely related to multi-step reasoning processes. LLOC, by comparison, is a coarser proxy that conflates structure with verbosity.

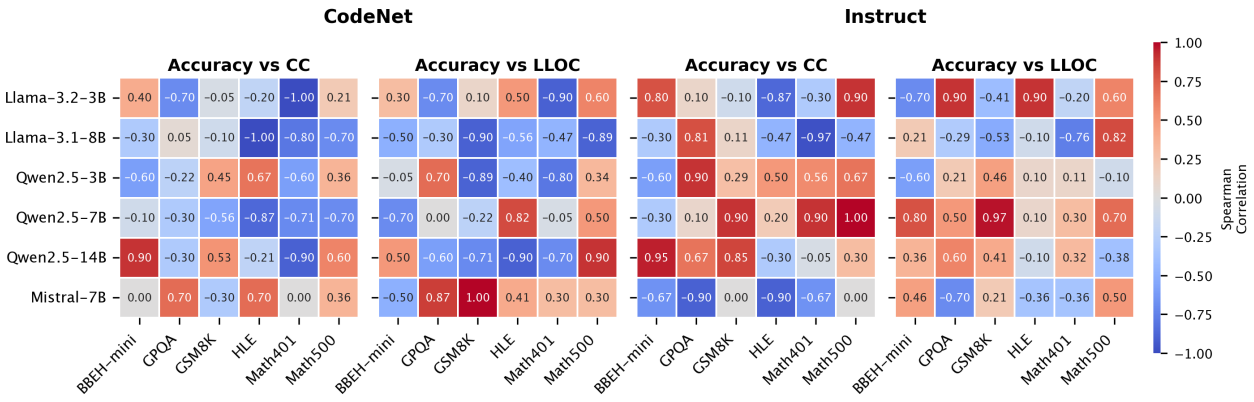

Figure 5: ***Correlation between code complexity and downstream reasoning accuracy.*** Spearman correlations between benchmark accuracy and training-data complexity level, computed across complexity-controlled dataset splits. Results are shown separately for solution-driven (CodeNet) and problem-driven (Instruct) settings, using cyclomatic complexity (CC) and logical lines of code (LLOC) as structural measures. *Correlation calculation is detailed in Appendix G.*

Absolute complexity partially aligns across dataset constructions. For CodeNet, all `Qwen` models tend to peak at the low CC split, whereas for Instruct, they peak at the max CC split. Despite this apparent discrepancy, both peaks correspond to a similar absolute CC value (≈ 10; Figure 4). This alignment suggests that absolute structural complexity–rather than whether that complexity arises from task requirements or solution variation–better explains the observed alignment than split labels alone. Dataset construction primarily determines *how* models encounter this complexity, but the effective complexity range itself appears to be model-specific and largely invariant across settings.

High structural complexity can degrade reasoning beyond an effective range. In several settings, fine-tuning on higher-complexity code reduces accuracy relative not only to other complexity ranges, but also to the NL baseline. For example, the `Llama` models exhibit strong negative correlations ($\rho \approx -1.00$) for CC splits in both CodeNet and Instruct settings (Figure 5), indicating that increasing structural complexity beyond a certain range can impair reasoning. These effects are also visible in the accuracy curves, which commonly dip at the highest complexity levels across models (Figure 4). These results point to a mismatch between structural complexity and model capacity: when code becomes overly complex – for example, containing unnecessary branching, deeply nested control flow, or redundant decision paths – it can introduce noise that obscures useful reasoning patterns. In such cases, additional complexity does not provide richer supervision, but instead leads to negative transfer, reducing the effectiveness of code as a reasoning signal.

Mistral exhibits qualitatively different behaviour. `Mistral-7B` displays a distinctive "U"-shaped performance curve across both CC-based datasets (Figure 3), benefiting most from either very simple or very complex code, with degraded performance at intermediate levels. Rather than contradicting the broader trend, this reinforces the central point that complexity effects are model-dependent: there is no single universally optimal complexity range. Different models appear to operate under different effective complexity regimes, suggesting that complexity-aware data selection should be calibrated to the target model rather than applied as a fixed global rule. We leave a deeper investigation of these model-specific effects to future work.

Fine-grained takeaway.

Across our settings, the effect of code complexity is not governed by split labels alone. The INSTRUCT setting shows stronger and more consistent trends than alternative solutions to the same task, and CC provides a more stable signal than LLOC. However, the effective range remains model-dependent: higher structural complexity can help in some cases, but can also degrade reasoning once it exceeds what a model appears to internalise effectively.

5 Discussion

Our results suggest that the relationship between code and reasoning is more structured–and more fragile–than is often assumed. Rather than acting as a uniformly beneficial training signal, low levels of code data appear to support reasoning only when the code's structural properties align with what a model can effectively internalise. Below, we briefly discuss what this implies for how code helps reasoning, and when it may fail to do so.

Why more code is not necessarily better. Previous work often emphasises that increasing the amount or diversity of code data will improve downstream reasoning performance (Aryabumi et al., 2024; Rozière et al., 2024; Wei et al., 2024). Our findings challenge this view: once structural complexity is controlled, additional or more complex code does not reliably lead to better reasoning, and can in some cases be actively harmful. This suggests that previously reported gains from large code corpora may depend not only on diversity, but also on exposure to particular structural properties of the code itself.

Structural complexity as implicit code chain-of-thought. Both natural-language chain-of-thought prompting (Wei et al., 2022b; Feng et al., 2023) and similar program-based techniques (Chen et al., 2023; Lin et al., 2025) improve reasoning by externalising intermediate structure and decomposing problems into explicit steps. This provides a natural way to interpret our results, where complex code supplies a similar scaffold through control flow and branching during fine-tuning. However, when structural complexity becomes too high, this scaffold can break down, introducing brittle control flow or optimisation difficulty that obscures rather than clarifies the reasoning signal.

Implications for data-centric reasoning improvement. Rather than relying on broad, mixed corpora, carefully selecting or constructing code with appropriate structural properties can yield stronger reasoning gains at fixed data or training budgets. This complements recent arguments that data quality and structure can rival or exceed gains from scale alone (Longpre et al., 2024), and is especially relevant in settings where high-quality code data is expensive to curate or fine-tune on (Chen et al., 2025). More broadly, our results suggest that improving reasoning through code is less about increasing exposure to programming in general, and more about identifying which computational structures best support multi-step reasoning in practice. A practical consequence is that code selection can be treated as a lightweight model-specific tuning problem: compute simple structural metrics over candidate code data, construct a small number of complexity-stratified subsets, run pilot fine-tunes, and then scale training within the empirically favourable range rather than defaulting to fully mixed corpora.

6 Limitations

This study has several limitations. First, our experiments focus on a small set of widely used programming languages and open-weight models; although these cover diverse coding styles, parameter scales, and model architectures, our findings may not fully generalise to other languages, proprietary models, or different training stages. Second, we characterise structure using two established static metrics–cyclomatic complexity and logical lines of code–which capture important aspects of control flow and program size; whilst well suited to an initial investigation, they do not exhaust the space of code properties that may influence reasoning. Third, the INSTRUCT setting combines datasets from different sources, so source-specific style, formatting, or generation effects may co-vary with structural complexity despite our balancing procedures. Finally,

our conclusions concern relative data-selection effects under LoRA fine-tuning on relatively small datasets (8,087 samples per split), which enables controlled comparisons but may not capture all behaviours that emerge at larger scales. Whether the same effective complexity ranges hold under full fine-tuning, continued pre-training, or substantially larger code mixtures remains an important direction for future work.

7 Conclusion

In this work, we studied how the structural properties of the code used during fine-tuning influence the reasoning abilities of LLMs. By systematically varying code complexity and length in both solution-driven and problem-driven settings, and evaluating across multiple model families and reasoning benchmarks, we show that code is not a uniform training signal for reasoning. Additionally, in 83% of experiments, restricting the fine-tuning data to an appropriate complexity or length range yields better downstream reasoning performance than training on a diverse mix of code. Our results highlight a data-centric path for improving reasoning–one that focusses on the *specific structural properties* of the code that is used for training.

Our study deliberately focusses on simple, interpretable structural metrics to enable controlled analysis, but this choice also points to several promising directions for future work. More expressive measures of code structure–such as hybrid metrics that combine control flow, data flow, and semantic patterns, or qualitative indicators of programming techniques with different structural patterns–may better capture the aspects of code that support reasoning. Exploring these directions may further clarify how code functions as an implicit form of chain-of-thought during training, and help translate structural insights into more targeted and efficient training data curation.

References

Amal Abed, Ivan Lukic, Jörg K. H. Franke, and Frank Hutter. Increasing LLM Coding Capabilities through Diverse Synthetic Coding Tasks. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, November 2025.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.298.

Viraat Aryabumi, Yixuan Su, Raymond Ma, Adrien Morisot, Ivan Zhang, Acyr Locatelli, Marzieh Fadaee, Ahmet Üstün, and Sara Hooker. To Code, or Not To Code? Exploring Impact of Code in Pre-training, August 2024.

Dan Biderman, Jacob Portes, Jose Javier Gonzalez Ortiz, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley, Jonathan Frankle, Cody Blakeney, and John Patrick Cunningham. LoRA learns less and forgets less. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856.

Laura Diana Cernau, Laura Diosan, and Camelia Serban. Unveiling Hybrid Cyclomatic Complexity: A Comprehensive Analysis and Evaluation as an Integral Feature in Automatic Defect Prediction Models, April 2025.

Meng Chen, Philip Arthur, Qianyu Feng, Cong Duy Vu Hoang, Yu-Heng Hong, Mahdi Kazemi Moghaddam, Omid Nezami, Duc Thien Nguyen, Gioacchino Tangari, Duy Vu, Thanh Vu, Mark Johnson, Krishnaram Kenthapadi, Don Dharmasiri, Long Duong, and Yuan-Fang Li. Mastering the Craft of Data Synthesis for CodeLLMs. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 12484–12500, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.620.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks, October 2023.

Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. ARC-AGI-2: A New Challenge for Frontier AI Reasoning Systems, May 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems, November 2021.

D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994. doi: 10.1109/2.303623.

Matt Cone. Extended Syntax. https://www.markdownguide.org/extended-syntax/, 2025.

Marc Dillmann, Julien Siebert, and Adam Trendowicz. Evaluation of large language models for assessing code maintainability, January 2024.

Yadolah Dodge. Spearman Rank Correlation Coefficient. In *The Concise Encyclopedia of Statistics*, pp. 502–505. Springer, New York, NY, 2008. ISBN 978-0-387-32833-1. doi: 10.1007/978-0-387-32833-1_379.

escomplex. Escomplex/escomplex. https://github.com/escomplex/escomplex, 2025.

Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. Towards revealing the mystery behind chain of thought: A theoretical perspective. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, pp. 70757–70798, Red Hook, NY, USA, December 2023. Curran Associates Inc.

Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, Inc., USA, 3rd edition, September 2014. ISBN 978-1-4398-3822-8.

Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic Complexity Density and Software Maintenance Productivity. *IEEE Trans. Software Eng.*, 17(12):1284–1288, 1991. doi: 10.1109/32.106988.

GitHub Staff. Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1, October 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. The Llama 3 Herd of Models, November 2024.

Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, April 1977. ISBN 978-0-444-00205-1.

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. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021) Track on Datasets and Benchmarks*. arXiv, November 2021. doi: 10.48550/arXiv.2103.03874.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*, October 2021.

Jie Huang and Kevin Chen-Chuan Chang. Towards Reasoning in Large Language Models: A Survey. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 1049–1065, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.67.

ise-uiuc. Ise-uiuc/Magicoder-Evol-Instruct-110K. https://huggingface.co/datasets/ise-uiuc/Magicoder-Evol-Instruct-110K, December 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, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B, October 2023.

Mehran Kazemi, Bahare Fatemi, Hritik Bansal, John Palowitch, Chrysovalantis Anastasiou, Sanket Vaibhav Mehta, Lalit K Jain, Virginia Aglietti, Disha Jindal, Peter Chen, Nishanth Dikkala, Gladys Tyen, Xin Liu, Uri Shalit, Silvia Chiappa, Kate Olszewska, Yi Tay, Vinh Q. Tran, Quoc V Le, and Orhan Firat. BIG-Bench Extra Hard. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26473–26501, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1285.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, pp. 22199–22213, Red Hook, NY, USA, November 2022. Curran Associates Inc. ISBN 978-1-7138-7108-8.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's Verify Step by Step. In *12th International Conference on Learning Representations (ICLR24)*, May 2023.

Honglin Lin, Qizhi Pei, Xin Gao, Zhuoshi Pan, Yu Li, Juntao Li, Conghui He, and Lijun Wu. Scaling Code-Assisted Chain-of-Thoughts and Instructions for Model Reasoning, October 2025.

Yang Liu, Jiahuan Cao, Chongyu Liu, Kai Ding, and Lianwen Jin. Datasets for large language models: A comprehensive survey. *Artificial Intelligence Review*, 58(12):403, October 2025. ISSN 1573-7462. doi: 10.1007/s10462-025-11403-7.

Shayne Longpre, Gregory Yauney, Emily Reif, Katherine Lee, Adam Roberts, Barret Zoph, Denny Zhou, Jason Wei, Kevin Robinson, David Mimno, and Daphne Ippolito. A Pretrainer's Guide to Training Data: Measuring the Effects of Data Age, Domain Coverage, Quality, & Toxicity. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3245–3276, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.naacl-long.179.

Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. An Empirical Study of Catastrophic Forgetting in Large Language Models During Continual Fine-tuning, January 2025.

Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. At Which Training Stage Does Code Data Help LLMs Reasoning?, September 2023.

Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pp. 407, Washington, DC, USA, October 1976. IEEE Computer Society Press.

meta-llama. Meta-llama/Llama-3.2-3B-Instruct · Hugging Face. https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct, December 2024.

mistralai. Mistralai/Mistral-7B-Instruct-v0.3 · Hugging Face. https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3, 2025.

Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry W. Boehm. A SLOC Counting Standard, 2007.

nickrosh. Nickrosh/Evol-Instruct-Code-80k-v1. https://huggingface.co/datasets/nickrosh/Evol-Instruct-Code-80k-v1, October 2024.

OpenAI. GPT-5 mini - API. https://platform.openai.com, 2025.

Long Phan, Alice Gatti, Ziwen Han, et al. Humanity's Last Exam, September 2025.

pmd. Pmd/pmd. https://github.com/pmd/pmd/releases, 2025.

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, August 2021.

Qwen. Qwen/Qwen2.5-14B-Instruct · Hugging Face. https://huggingface.co/Qwen/Qwen2.5-14B-Instruct, December 2025a.

Qwen. Qwen/Qwen2.5-3B-Instruct · Hugging Face. https://huggingface.co/Qwen/Qwen2.5-3B-Instruct, December 2025b.

Qwen. Qwen/Qwen2.5-7B-Instruct · Hugging Face. https://huggingface.co/Qwen/Qwen2.5-7B-Instruct, December 2025c.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 Technical Report, January 2025.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A Graduate-Level Google-Proof Q&A Benchmark, November 2023.

rombodawg. code_instruct_alpaca_vicuna_wizardlm_56k_backup. https://huggingface.co/datasets/rombodawg/code_instruct_alpaca_vicuna_wizardlm_56k_backup, 2023.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, 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, January 2024.

rubik. Rubik/radon. https://github.com/rubik/radon, 2025.

Melika Sepidband, Hamed Taherkhani, Song Wang, and Hadi Hemmati. Enhancing LLM-Based Code Generation with Complexity Metrics: A Feedback-Driven Approach. In *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1416–1426. IEEE Computer Society, July 2025. ISBN 979-8-3315-7434-5. doi: 10.1109/COMPSAC65507.2025.00178.

V. Siahaan, H. Ginting, and M. Amri. Cyclomatic Complexity and Maintainability in Modern Software: A Systematic Review. *Journal of Data Science, Technology, and Computer Science*, 5(1):8–16, June 2025. ISSN 2809-171X. doi: 10.63703/distances.v5i1.83.

Masoud Jamshidiyan Tehrani and Sattar Hashemi. Assessing Vulnerability in Smart Contracts: The Role of Code Complexity Metrics in Security Analysis, March 2025.

Saurabh Tiwari and Santosh Singh Rathore. Coupling and Cohesion Metrics for Object-Oriented Software: A Systematic Mapping Study. In *Proceedings of the 11th Innovations in Software Engineering Conference*, ISEC '18, pp. 1–11, New York, NY, USA, February 2018. Association for Computing Machinery. ISBN 978-1-4503-6398-3. doi: 10.1145/3172871.3172878.

Luong Trung, Xinbo Zhang, Zhanming Jie, Peng Sun, Xiaoran Jin, and Hang Li. ReFT: Reasoning with Reinforced Fine-Tuning. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7601–7614, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.410.

Abdul Waheed, Zhen Wu, Carolyn Rosé, and Daphne Ippolito. On Code-Induced Reasoning in LLMs. In *14th International Conference on Learning Representations (ICLR26)*. arXiv, 2026. doi: 10.48550/arXiv.2509.21499.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-Consistency Improves Chain of Thought Reasoning in Language Models, March 2023.

Yiping Wang, Qing Yang, Zhiyuan Zeng, Liliang Ren, Liyuan Liu, Baolin Peng, Hao Cheng, Xuehai He, Kuan Wang, Jianfeng Gao, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. Reinforcement Learning for Reasoning in Large Language Models with One Training Example. In *39th Conference on Neural Information Processing Systems (NeurIPS 2025)*, 2025.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent Abilities of Large Language Models. *Transactions on Machine Learning Research*, 2022a.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, pp. 24824–24837, Red Hook, NY, USA, November 2022b. Curran Associates Inc. ISBN 978-1-7138-7108-8.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-INSTRUCT. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *ICML'24*, pp. 52632–52657, Vienna, Austria, July 2024. JMLR.org.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and Julian McAuley. Code to Think, Think to Code: A Survey on Code-Enhanced Reasoning and Reasoning-Driven Code Intelligence in LLMs, February 2025.

Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, and Songfang Huang. How well do Large Language Models perform in Arithmetic tasks?, March 2023.

Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. When Scaling Meets LLM Finetuning: The Effect of Data, Model and Finetuning Method. In *12th International Conference on Learning Representations (ICLR24)*. arXiv, February 2024a. doi: 10.48550/arXiv.2402.17193.

Xinlu Zhang, Zhiyu Zoey Chen, Xi Ye, Xianjun Yang, Lichang Chen, William Yang Wang, and Linda Ruth Petzold. Unveiling the Impact of Coding Data Instruction Fine-Tuning on Large Language Models Reasoning. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*. arXiv, December 2024b. doi: 10.1609/aaai.v39i24.34789.

A Complexity Metrics Implementation Details

To characterise the complexity of the code used during fine-tuning, we compute two complementary static metrics over all samples: cyclomatic complexity (CC) and logical lines of code (LLOC). This appendix reports the implementation details of their calculation.

A.1 Complexity Metric Calculation

We calculate our complexity metrics using established open-source static analysis tools, here we provide details on our implementation of these tools. For both datasets, we first ensure that we have the raw solution code extracted from the natural language response: for CodeNet, the code is already provided separately; for Instruct, the natural language responses are given in `Markdown` format, therefore we can use `regex` matching to extract code blocks and programming languages by searching for a triple backtick followed by a programming language name (Cone, 2025). We then process the code for each language (`Python`, `JavaScript`, `Java`) separately:

- For `Python`, we install and use the `Radon` library[3]. We use the `radon.complexity.cc_visit` function to calculate CC; and the `radon.raw.analyze` function to calculate LLOC.
- For `JavaScript`, we use the `complexity-report`[4] command-line tool, to run the `escomplex` library[5] against our code. We use the `cr` command with the `-format json` option, to produce a report that contains both CC and LLOC.
- For `Java`, we use the `PMD` static code analyzer[6], to be run from the command line. We use the `pmd` command with the `-f json` option, and passing in a ruleset that requests `CyclomaticComplexity` and `NcssCount`, to produce a report that contains both CC and LLOC.

A.2 Complexity Metric Aggregation

Cyclomatic complexity is defined at the level of individual functions; following common practice in software metrics, we aggregate function-level values to the solution level by taking the maximum over all functions included. This reflects the intuition that structural complexity is often dominated by the most complex execution path (Gill & Kemerer, 1991). By contrast, LLOC acts as an additive size-based measure and therefore does not require special handling–we simply sum all logical lines of code contained in the solution.

B CodeNet Augmentation Prompts

To convert Project CodeNet (Puri et al., 2021) into an instruction–response format suitable for supervised fine-tuning, we apply an LLM-based augmentation step using a fixed system prompt and two task-specific user prompts. All prompts are applied to each CodeNet problem–solution pair to ensure consistency across augmented samples. We generated responses to the prompts using the `gpt-5-mini-2025-08-07` model (OpenAI, 2025) via the OpenAI API[7], with the default API parameters (`reasoning.effort = medium` and `text.verbosity = medium`).

System prompt. All augmentation calls use the same system prompt, which constrains the model to return only the requested content without additional commentary:

```
You are a helpful assistant that will assist in creating a new code-based benchmark
dataset.  When responding, you only provide exactly what is requested, with no
additional text.
```

[3]Radon: https://pypi.org/project/radon/
[4]complexity-report: https://www.npmjs.com/package/complexity-report
[5]escomplex: https://www.npmjs.com/package/escomplex
[6]PMD: https://pmd.github.io/
[7]OpenAI API: https://openai.com/api/

Instruction-template prompt. To convert HTML problem statements into natural-language instructions, we prompt the model to produce a concise, language-agnostic instruction that preserves the original task specification exactly and includes a `<language>` placeholder to be substituted later. A single instruction template is generated per CodeNet record.

```
I am augmenting the Project CodeNet (by IBM) dataset, converting it into an
instruction / response dataset that can be used for supervised finetuning, and I
need assistance.
The current problem statements are provided in HTML, and I need you to convert them
into a natural language prompt instruction that I can use to ask models to generate
code.
The instruction must be programming language agnostic, but you must provide a
<language> token in the instruction, that I can replace with the programming
language that must be used.
It is vital that the requested specifications are exactly the same as the original.
Only provide the instruction exactly as it should be used in the dataset.  Here is
the original HTML problem statement:
{html}
```

Response-template prompt. To standardise model outputs, we prompt the model to generate a natural-sounding response template that wraps the solution code in surrounding explanatory text. The template is language-agnostic and contains two placeholders: `<language>` for the programming language and `<code>` for the solution code, which is inserted verbatim during dataset construction. Three different response templates are generated per CodeNet record, so that each programming language has a unique response template.

```
I am augmenting the Project CodeNet (by IBM) dataset, converting it into an
instruction / response dataset that can be used for supervised finetuning, and I
need assistance.
The solutions are currently provided as just raw code, and I need your help to turn
them into readable and useful model responses that can be used for training.
I need you to provide a template for a response that would read naturally to a user,
you should add the surrounding text that LLMs typically provide, reading as if it is
a real response from an LLM solving the task.  Do not include specifics of the code,
approach or algorithm though - you have not seen the code yet, so it might not be
accurate.
The response should be language agnostic (do not use those words though), but must
contain a <language> token, that I can replace with the programming language that is
used.
You must also provide a <code> token that I will replace with the code block of the
response, there is no need for a corresponding </code>.
The <code> token must be surrounded by newlines, but I will handle correctly having
the code itself contained within triple backticks (as per markdown).
Only provide the response template exactly as it should be used in the dataset.
Here is the instruction:
{instruction}
```

C Dataset Statistics

This appendix reports additional statistics for the datasets used in our fine-tuning experiments. We create a solution-driven complexity dataset (CodeNet) and a problem-driven complexity dataset (Instruct), each of which has twelve splits: five different complexity levels (min, low, mid, high, max) and a control (ctrl) for each complexity metric (CC and LLOC).

Table 1: ***Structural complexity statistics.*** We report mean cyclomatic complexity (CC) and logical lines of code (LLOC) for each split of the CODENET (solution-driven complexity) and INSTRUCT (problem-driven complexity) datasets.

Split	CodeNet: solution-driven complexity				Instruct: problem-driven complexity			
	CC dataset split		LLOC dataset split		CC dataset split		LLOC dataset split	
	Avg. CC	Avg. LLOC	Avg. CC	Avg. LLOC	Avg. CC	Avg. LLOC	Avg. CC	Avg. LLOC
MIN	7.79	37.67	8.83	32.67	0.63	8.35	0.96	3.43
LOW	9.91	44.65	10.66	40.75	0.69	8.26	1.30	4.74
MID	13.99	58.81	14.32	57.69	2.00	10.76	2.62	9.65
HIGH	20.63	82.50	19.95	84.82	3.89	15.25	4.01	16.06
MAX	43.03	169.67	40.63	180.17	11.12	29.83	7.58	43.94
CTRL	18.84	76.14	19.21	81.83	3.78	15.02	3.78	15.81

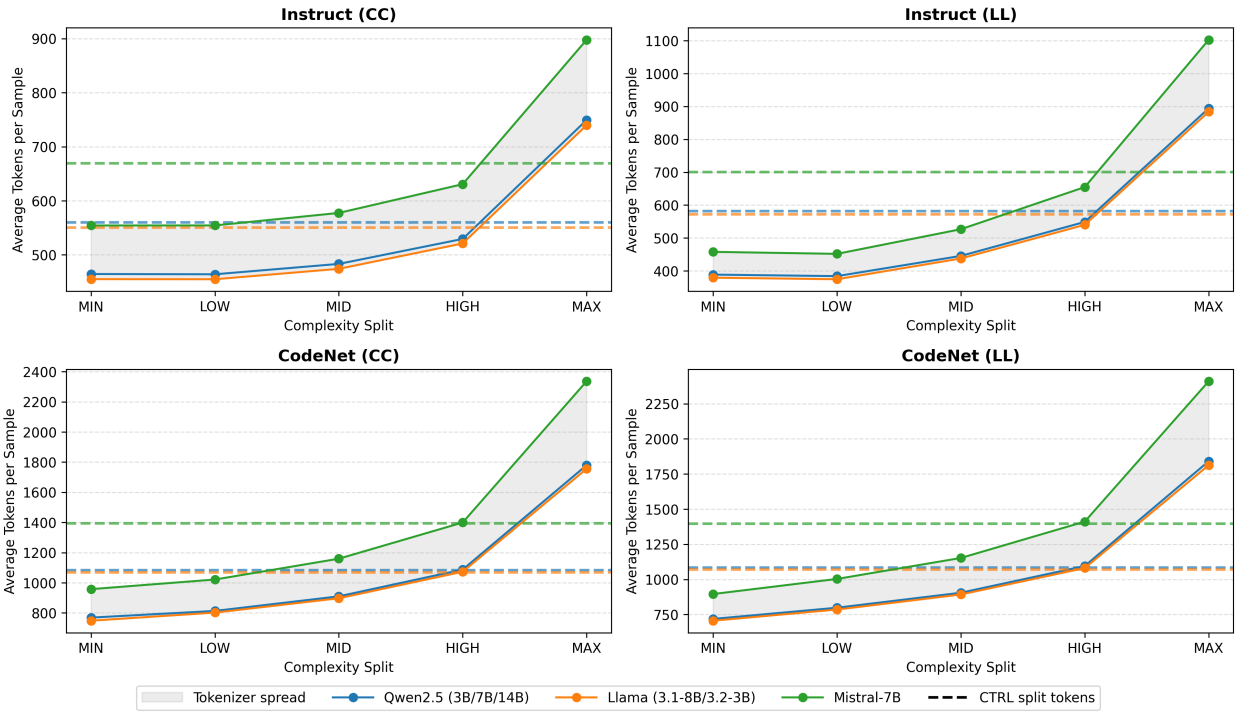

Figure 6: **Average token count per complexity split across datasets and models.** Average sequence length for each complexity bucket (MIN–MAX) across INSTRUCT and CODENET. Dashed lines show the corresponding CTRL token averages.

C.1 Programming Language Statistics

All splits in the CODENET dataset contain 2,919 `Python` samples, 1,890 `JavaScript` samples and 3,278 `Java` samples. All splits in the INSTRUCT dataset contain 3,688 `Python` samples, 2,789 `JavaScript` samples and 1,610 `Java` samples.

C.2 Metric Values Across Complexity Splits

Table 1 reports the mean CC and LLOC values for each split across both datasets. Reporting both metrics is important because CC and LLOC are correlated in practice; including LLOC helps control for confounding effects of code length when assessing structural complexity.

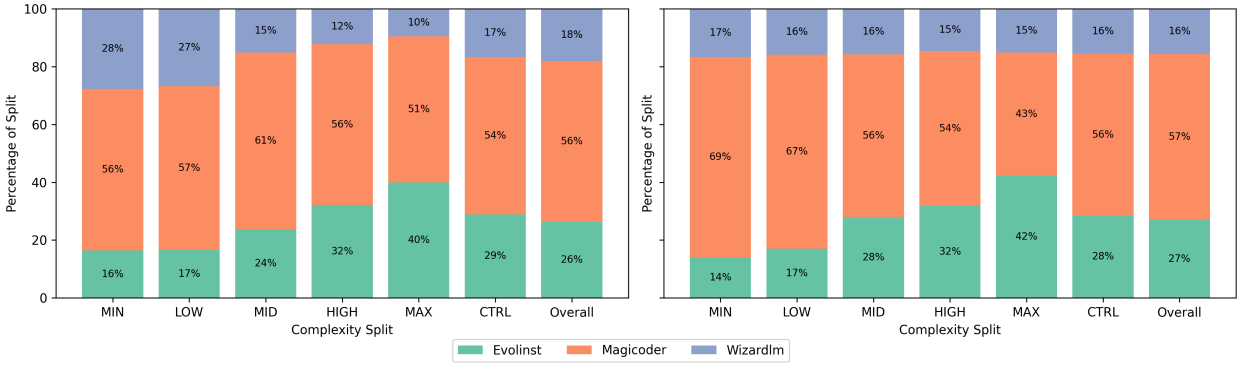

Figure 7: **Instruct dataset composition across complexity splits.** Proportion of samples from `Evol-Instruct`, `Magicoder`, and `WizardLM` across CC and LLOC splits of the INSTRUCT dataset.

C.3 Token Length Across Complexity Splits

Figure 6 shows the average token count per sample across datasets and complexity splits. This is important to consider when interpreting results, as higher complexity is associated with longer sequences. However, this correlation is moderate and does not appear sufficient to explain the observed effects. In particular, the MIN–MID range shows similar token counts but different reasoning performance, indicating improvements are not driven by length alone. The largest increase in length occurs only at MAX, while key performance trends emerge earlier.

C.4 Instruct Seed Dataset Composition

Our INSTRUCT dataset is constructed from three seed datasets: `Evol-Instruct`, `Magicoder`, and `WizardLM`. We ensure an even spread of programming languages across splits, while prioritising a balanced distribution of structural complexity rather than exact proportions from each source. Figure 7 shows the dataset composition. While proportions vary slightly, they remain approximately balanced, reducing–but not eliminating–the possibility that source composition drives the observed trends. We believe this controlled mixing provides a stronger comparison than uneven complexity distributions.

D Model Evaluation Details.

D.1 Model Details

Table 2 reports the full details of all models used in our experiments. We document parameter scale, context window, provider, and knowledge cut-off to make transparent the architectural and training differences across models and to facilitate reproducibility of our results. All models are publicly available through Hugging Face[8], and we use only officially released instruct variants to ensure consistent evaluation behaviour across tasks. All models employ grouped-query attention (Ainslie et al., 2023); therefore attention heads are reported as the number of query heads and key–value heads (Q/KV).

D.2 Training configuration

We fine-tune all models using Low-Rank Adaptation (LoRA) (Hu et al., 2021). While we generally follow standard practices, we specifically set the LoRA rank $r = 16$, alpha $\alpha = 16$, and dropout to 0. We apply LoRA adapters to all linear modules, including `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`. We use the AdamW optimizer (`adamw_torch`) with a learning rate of 2×10^{-5}, a cosine learning rate scheduler, and a warm-up ratio of 0.1. Training is conducted for 2 epochs with a per-device batch size

[8]Hugging Face: https://huggingface.co/

Table 2: ***Full details for all models evaluated in our experiments.*** We report model size, context length, and architectural characteristics for each model.

Model Name	Full Model Version	Parameters	Context Length	Attention Heads (Q/KV)	Provider	Knowledge Cutoff	Source
`Qwen2.5-3B`	`Qwen2.5-3B-Instruct`	3.09B	32K	16 / 2	Alibaba	Dec. 2023	Hugging Face (Qwen, 2025b)
`Qwen2.5-7B`	`Qwen2.5-7B-Instruct`	7.61B	32K	28 / 4	Alibaba	Dec. 2023	Hugging Face (Qwen, 2025c)
`Qwen2.5-14B`	`Qwen2.5-14B-Instruct`	14.70B	128K	40 / 8	Alibaba	Dec. 2023	Hugging Face (Qwen, 2025a)
`Llama-3.2-3B`	`Llama-3.2-3B-Instruct`	3.21B	128K	24 / 8	Meta	Dec. 2023	Hugging Face (meta-llama, 2024)
`Llama-3.1-8B`	`Llama-3.1-8B-Instruct`	8.00B	128K	32 / 8	Meta	Dec. 2023	Hugging Face (meta-llama, 2024)
`Mistral-7B`	`Mistral-7B-Instruct-v0.3`	7.00B	32K	32 / 8	Mistral	May 2024	Hugging Face (mistralai, 2025)

of 4 and 4 gradient accumulation steps, resulting in an effective batch size of 16. All models are trained with a maximum sequence length of 32,768 tokens using `bfloat16` precision.

D.3 Evaluation configuration

We maximize reproducibility by using a greedy decoding strategy with a temperature of 0.0, top-p of 1.0, and a maximum generation length of 16,384 tokens. To standardize input formats across models, we append the following suffix to every query: "`\n Please reason step by step, and put your final answer within \boxed{}.`". We extract the final answer by parsing the content within the `\boxed{}` delimiters or other standard indicators (e.g., `<answer>`, `###`). To ensure robust evaluation, we utilize the `math_verify` library[9] to match the extracted answers against the ground truth.

E Comprehensive Results

Table 3 reports the complete set of per-benchmark results for all models and training configurations considered in this study. For each benchmark, we include accuracies obtained after fine-tuning on complexity-controlled code datasets under both the solution-driven (CodeNet) and problem-driven (Instruct) settings, split by cyclomatic complexity (CC) and logical lines of code (LLOC), alongside the corresponding control (ctrl) datasets and the natural language (NL) baseline. This table is provided for completeness and to support detailed inspection of individual model–benchmark behaviours.

F Does More Mixed-Complexity Data Remove the Effect?

To test whether targeted complexity splits remain beneficial when compared against substantially more training data, we conduct an additional experiment using the full Instruct CC mixture. This dataset combines all five CC buckets, giving approximately 40k samples, compared with approximately 8k samples in each individual complexity split. We fine-tune two representative models using the same training configuration as in the main experiments, and compare full-dataset training against the best-performing individual complexity bucket for each benchmark.

Table 4 shows that targeted complexity buckets outperform the larger mixed dataset for both tested models. For `Qwen2.5-7B`, the best individual bucket reaches 41.07% average accuracy compared with 36.18% for full-dataset training ($\Delta = -4.89\%$), with particularly large gains on GSM8K (94.00% vs. 83.47%) and Math500 (60.40% vs. 47.20%). For `Llama-3.2-3B`, the average difference is smaller (28.74% vs. 28.08%; $\Delta = -0.66\%$), but the best bucket still outperforms the full dataset on every benchmark.

These results reinforce our main finding: targeted complexity selection can be more effective than training on mixed-complexity data, even when the mixed dataset is substantially larger and contains the best-performing subset. This suggests that increasing data scale alone does not necessarily improve reasoning performance; in some settings, mixing across structural regimes may dilute the signal provided by the most useful complexity range.

[9]Math-Verify: https://github.com/huggingface/math-verify

Table 3: ***Full per-benchmark results for all models.*** Average accuracy on each reasoning benchmark following fine-tuning on complexity-controlled code subsets with 95% bootstrap confidence intervals. We report results for fine-tuning on solution-driven (CODENET) and problem-driven (INSTRUCT) datasets, split by cyclomatic complexity (CC) and logical lines of code (LLOC), together with control splits and the natural language (NL) baseline, where the model is fine-tuned on a strictly non-code dataset.

	Model	Baseline	CodeNet: CC split						CodeNet: LLOC split					
		NL	MIN	LOW	MID	HIGH	MAX	CTRL	MIN	LOW	MID	HIGH	MAX	CTRL
BBEH-mini	Qwen2.5-3B	6.1 ± 2.2	8.9 ± 2.6	11.5 ± 2.9	9.6 ± 2.7	7.8 ± 2.5	8.5 ± 2.5	9.6 ± 2.7	10.2 ± 2.8	10.0 ± 2.7	8.7 ± 2.6	10.4 ± 2.8	10.0 ± 2.7	10.4 ± 2.8
	Qwen2.5-7B	9.3 ± 2.7	10.4 ± 2.8	12.2 ± 3.0	9.8 ± 2.7	10.0 ± 2.7	11.5 ± 2.9	9.6 ± 2.7	12.0 ± 3.0	11.5 ± 2.9	10.4 ± 2.8	10.9 ± 2.8	10.7 ± 2.8	10.7 ± 2.8
	Qwen2.5-14B	12.8 ± 3.1	12.0 ± 3.0	12.6 ± 3.0	14.6 ± 3.2	13.7 ± 3.1	14.8 ± 3.2	13.0 ± 3.1	12.4 ± 3.0	14.1 ± 3.2	12.6 ± 3.0	15.7 ± 3.3	13.9 ± 3.2	13.5 ± 3.1
	Llama-3.2-3B	0.4 ± 0.6	3.5 ± 1.7	2.6 ± 1.5	2.8 ± 1.5	3.0 ± 1.6	3.9 ± 1.8	3.0 ± 1.6	3.0 ± 1.6	2.0 ± 1.3	2.6 ± 1.5	2.4 ± 1.4	4.3 ± 1.9	4.1 ± 1.8
	Llama-3.1-8B	3.0 ± 1.6	7.4 ± 2.4	6.3 ± 2.2	6.7 ± 2.3	8.5 ± 2.5	5.9 ± 2.1	7.0 ± 2.3	7.2 ± 2.4	7.4 ± 2.4	7.8 ± 2.5	5.9 ± 2.1	6.3 ± 2.2	6.5 ± 2.3
	Mistral-7B	5.2 ± 2.0	5.7 ± 2.1	5.2 ± 2.0	4.8 ± 2.0	4.6 ± 1.9	6.3 ± 2.2	6.3 ± 2.2	5.9 ± 2.1	5.0 ± 2.0	5.7 ± 2.1	4.6 ± 1.9	5.2 ± 2.0	5.2 ± 2.0
GPQA	Qwen2.5-3B	17.4 ± 3.5	19.4 ± 3.7	19.9 ± 3.7	20.8 ± 3.8	19.4 ± 3.7	19.4 ± 3.7	20.5 ± 3.7	19.4 ± 3.7	20.1 ± 3.7	21.2 ± 3.8	20.8 ± 3.8	21.0 ± 3.8	21.0 ± 3.8
	Qwen2.5-7B	16.7 ± 3.5	21.2 ± 3.8	23.9 ± 3.9	22.3 ± 3.9	23.0 ± 3.9	20.8 ± 3.8	25.0 ± 4.0	21.9 ± 3.8	22.3 ± 3.9	22.8 ± 3.9	24.8 ± 4.0	20.3 ± 3.7	21.2 ± 3.8
	Qwen2.5-14B	15.4 ± 3.3	25.4 ± 4.0	28.6 ± 4.2	26.8 ± 4.1	27.9 ± 4.2	25.0 ± 4.0	25.2 ± 4.0	26.8 ± 4.1	29.2 ± 4.2	29.7 ± 4.2	25.7 ± 4.0	24.6 ± 4.0	26.6 ± 4.1
	Llama-3.2-3B	14.7 ± 3.3	16.3 ± 3.4	15.4 ± 3.3	17.0 ± 3.5	14.5 ± 3.3	13.8 ± 3.2	12.7 ± 3.1	14.7 ± 3.3	15.8 ± 3.4	15.4 ± 3.3	14.5 ± 3.3	12.3 ± 3.0	12.7 ± 3.1
	Llama-3.1-8B	14.3 ± 3.2	14.3 ± 3.2	15.8 ± 3.4	15.6 ± 3.4	15.4 ± 3.3	15.4 ± 3.3	13.2 ± 3.1	17.2 ± 3.5	12.9 ± 3.1	16.3 ± 3.4	12.3 ± 3.0	16.5 ± 3.4	17.9 ± 3.5
	Mistral-7B	9.6 ± 2.7	5.1 ± 2.0	6.2 ± 2.2	5.8 ± 2.2	6.0 ± 2.2	7.6 ± 2.5	6.2 ± 2.2	6.0 ± 2.2	4.9 ± 2.0	6.5 ± 2.3	6.5 ± 2.3	7.1 ± 2.4	6.0 ± 2.2
GSM8K	Qwen2.5-3B	81.9 ± 5.3	88.0 ± 4.5	88.0 ± 4.5	86.0 ± 4.8	88.0 ± 4.5	89.0 ± 4.3	87.0 ± 4.7	88.5 ± 4.4	88.0 ± 4.5	88.0 ± 4.5	88.0 ± 4.5	86.5 ± 4.7	88.5 ± 4.4
	Qwen2.5-7B	85.3 ± 4.9	94.5 ± 3.2	94.0 ± 3.3	93.5 ± 3.4	93.0 ± 3.5	94.0 ± 3.3	94.0 ± 3.3	94.0 ± 3.3	94.0 ± 3.3	94.5 ± 3.2	93.5 ± 3.4	94.0 ± 3.3	94.0 ± 3.3
	Qwen2.5-14B	90.4 ± 4.1	96.5 ± 2.5	97.0 ± 2.4	97.0 ± 2.4	96.5 ± 2.5	97.5 ± 2.2	98.0 ± 1.9	96.5 ± 2.5	96.5 ± 2.5	97.0 ± 2.4	96.5 ± 2.5	95.0 ± 3.0	96.5 ± 2.5
	Llama-3.2-3B	62.3 ± 6.7	75.5 ± 6.0	75.5 ± 6.0	77.0 ± 5.8	78.0 ± 5.7	72.0 ± 6.2	80.0 ± 5.5	74.0 ± 6.1	76.5 ± 5.9	76.0 ± 5.9	79.5 ± 5.6	74.0 ± 6.1	80.0 ± 5.5
	Llama-3.1-8B	78.0 ± 5.7	85.5 ± 4.9	87.0 ± 4.7	89.0 ± 4.3	86.0 ± 4.8	85.5 ± 4.9	88.5 ± 4.4	88.5 ± 4.4	89.5 ± 4.2	88.0 ± 4.5	86.5 ± 4.7	84.0 ± 5.1	89.0 ± 4.3
	Mistral-7B	5.0 ± 3.0	60.0 ± 6.8	32.5 ± 6.5	28.0 ± 6.2	28.5 ± 6.3	45.5 ± 6.9	38.0 ± 6.7	27.0 ± 6.2	28.5 ± 6.3	41.0 ± 6.8	42.0 ± 6.8	45.5 ± 6.9	30.0 ± 6.4
HLE	Qwen2.5-3B	2.0 ± 0.6	1.6 ± 0.5	1.6 ± 0.5	1.9 ± 0.5	1.7 ± 0.5	1.7 ± 0.5	1.7 ± 0.5	1.9 ± 0.6	1.7 ± 0.5	1.6 ± 0.5	1.6 ± 0.5	1.9 ± 0.5	1.7 ± 0.5
	Qwen2.5-7B	2.0 ± 0.6	3.0 ± 0.7	3.1 ± 0.7	2.9 ± 0.7	2.9 ± 0.7	2.8 ± 0.7	2.9 ± 0.7	2.6 ± 0.6	2.7 ± 0.6	2.6 ± 0.6	2.9 ± 0.7	3.0 ± 0.7	2.4 ± 0.6
	Qwen2.5-14B	1.9 ± 0.5	1.7 ± 0.5	1.9 ± 0.5	1.7 ± 0.5	2.2 ± 0.6	1.5 ± 0.5	1.7 ± 0.5	1.8 ± 0.5	1.8 ± 0.5	1.7 ± 0.5	1.6 ± 0.5	1.7 ± 0.5	1.6 ± 0.5
	Llama-3.2-3B	0.3 ± 0.2	2.3 ± 0.6	2.7 ± 0.7	2.2 ± 0.6	2.0 ± 0.6	2.4 ± 0.6	2.3 ± 0.6	2.0 ± 0.6	2.4 ± 0.6	1.9 ± 0.5	2.2 ± 0.6	2.5 ± 0.6	2.3 ± 0.6
	Llama-3.1-8B	2.2 ± 0.6	2.4 ± 0.6	2.3 ± 0.6	2.1 ± 0.6	2.0 ± 0.6	1.9 ± 0.6	2.2 ± 0.6	2.7 ± 0.6	2.7 ± 0.6	2.7 ± 0.7	2.3 ± 0.6	2.4 ± 0.6	2.0 ± 0.6
	Mistral-7B	0.3 ± 0.2	0.7 ± 0.3	1.1 ± 0.4	0.8 ± 0.3	0.8 ± 0.4	1.7 ± 0.5	0.7 ± 0.3	0.9 ± 0.4	0.6 ± 0.3	0.9 ± 0.4	0.8 ± 0.4	1.3 ± 0.5	0.9 ± 0.4
Math401	Qwen2.5-3B	57.4 ± 4.8	59.4 ± 4.8	60.6 ± 4.8	59.6 ± 4.8	58.6 ± 4.8	59.1 ± 4.8	58.6 ± 4.8	59.6 ± 4.8	59.9 ± 4.8	59.1 ± 4.8	59.4 ± 4.8	58.6 ± 4.8	59.1 ± 4.8
	Qwen2.5-7B	57.9 ± 4.8	61.3 ± 4.8	61.3 ± 4.8	61.3 ± 4.8	61.3 ± 4.8	61.1 ± 4.8	61.6 ± 4.8	61.6 ± 4.8	62.3 ± 4.7	61.3 ± 4.8	60.6 ± 4.8	62.3 ± 4.7	62.6 ± 4.7
	Qwen2.5-14B	64.1 ± 4.7	64.3 ± 4.7	65.1 ± 4.7	62.8 ± 4.7	61.3 ± 4.8	60.1 ± 4.8	61.1 ± 4.8	65.1 ± 4.7	63.6 ± 4.7	65.3 ± 4.7	62.3 ± 4.7	60.6 ± 4.8	62.8 ± 4.7
	Llama-3.2-3B	45.9 ± 4.9	57.6 ± 4.8	56.9 ± 4.8	56.4 ± 4.9	55.9 ± 4.9	55.6 ± 4.9	56.9 ± 4.8	57.9 ± 4.8	57.6 ± 4.8	56.4 ± 4.9	55.6 ± 4.9	55.9 ± 4.9	55.1 ± 4.9
	Llama-3.1-8B	50.6 ± 4.9	53.9 ± 4.9	53.4 ± 4.9	53.6 ± 4.9	52.9 ± 4.9	53.1 ± 4.9	51.6 ± 4.9	53.6 ± 4.9	52.4 ± 4.9	51.9 ± 4.9	53.6 ± 4.9	51.9 ± 4.9	54.4 ± 4.9
	Mistral-7B	0.0 ± 0.1	9.0 ± 2.8	8.0 ± 2.7	7.7 ± 2.6	7.0 ± 2.5	11.5 ± 3.1	6.2 ± 2.4	8.5 ± 2.7	7.0 ± 2.5	8.0 ± 2.7	7.7 ± 2.6	10.0 ± 2.9	7.0 ± 2.5
Math500	Qwen2.5-3B	52.8 ± 4.4	50.8 ± 4.4	51.2 ± 4.4	49.4 ± 4.4	50.8 ± 4.4	52.4 ± 4.4	51.0 ± 4.4	51.0 ± 4.4	51.0 ± 4.4	51.2 ± 4.4	51.4 ± 4.4	51.0 ± 4.4	51.4 ± 4.4
	Qwen2.5-7B	46.8 ± 4.4	60.6 ± 4.3	60.2 ± 4.3	61.2 ± 4.3	60.0 ± 4.3	58.8 ± 4.3	60.2 ± 4.3	59.4 ± 4.3	60.0 ± 4.3	59.6 ± 4.3	61.0 ± 4.3	59.8 ± 4.3	61.2 ± 4.3
	Qwen2.5-14B	61.6 ± 4.3	63.4 ± 4.2	63.2 ± 4.2	63.0 ± 4.2	63.8 ± 4.2	64.6 ± 4.2	62.0 ± 4.3	63.0 ± 4.2	64.0 ± 4.2	63.6 ± 4.2	64.6 ± 4.2	64.8 ± 4.2	63.0 ± 4.2
	Llama-3.2-3B	26.6 ± 3.9	33.2 ± 4.1	32.6 ± 4.1	32.0 ± 4.1	32.6 ± 4.1	34.4 ± 4.2	33.4 ± 4.1	31.4 ± 4.1	32.0 ± 4.1	36.6 ± 4.2	34.2 ± 4.2	33.2 ± 4.1	32.0 ± 4.1
	Llama-3.1-8B	33.8 ± 4.1	39.0 ± 4.3	38.0 ± 4.3	37.6 ± 4.2	38.4 ± 4.3	37.0 ± 4.2	37.2 ± 4.2	37.8 ± 4.3	37.2 ± 4.2	37.2 ± 4.2	37.2 ± 4.2	36.8 ± 4.2	36.6 ± 4.2
	Mistral-7B	2.2 ± 1.3	8.8 ± 2.5	9.8 ± 2.6	9.8 ± 2.6	8.4 ± 2.4	10.8 ± 2.7	13.0 ± 2.9	7.8 ± 2.4	10.0 ± 2.6	10.6 ± 2.7	10.2 ± 2.7	9.2 ± 2.5	9.0 ± 2.5

	Model	Baseline	Instruct: CC split						Instruct: LLOC split					
		NL	MIN	LOW	MID	HIGH	MAX	CTRL	MIN	LOW	MID	HIGH	MAX	CTRL
BBEH-mini	Qwen2.5-3B	6.1 ± 2.2	8.3 ± 2.5	7.4 ± 2.4	6.7 ± 2.3	7.0 ± 2.3	7.2 ± 2.4	7.2 ± 2.4	7.8 ± 2.5	8.9 ± 2.6	7.4 ± 2.4	6.5 ± 2.3	7.6 ± 2.4	6.7 ± 2.3
	Qwen2.5-7B	9.3 ± 2.7	10.7 ± 2.8	9.1 ± 2.6	6.5 ± 2.3	8.7 ± 2.6	9.6 ± 2.7	10.0 ± 2.7	7.8 ± 2.5	7.0 ± 2.3	8.0 ± 2.5	11.5 ± 2.9	9.3 ± 2.7	8.7 ± 2.6
	Qwen2.5-14B	12.8 ± 3.1	12.4 ± 3.0	12.6 ± 3.0	12.6 ± 3.0	13.3 ± 3.1	13.3 ± 3.1	12.6 ± 3.0	12.2 ± 3.0	14.3 ± 3.2	13.5 ± 3.1	13.5 ± 3.1	13.9 ± 3.2	14.1 ± 3.2
	Llama-3.2-3B	0.4 ± 0.6	2.2 ± 1.3	2.8 ± 1.5	2.4 ± 1.4	3.3 ± 1.6	3.0 ± 1.6	2.8 ± 1.5	3.3 ± 1.6	1.7 ± 1.2	3.0 ± 1.6	2.2 ± 1.3	1.5 ± 1.1	3.0 ± 1.6
	Llama-3.1-8B	3.0 ± 1.6	7.2 ± 2.4	7.8 ± 2.5	6.7 ± 2.3	8.3 ± 2.5	6.3 ± 2.2	8.7 ± 2.6	5.7 ± 2.1	7.8 ± 2.5	5.7 ± 2.1	7.4 ± 2.4	6.7 ± 2.3	7.2 ± 2.4
	Mistral-7B	5.2 ± 2.0	5.2 ± 2.0	4.6 ± 1.9	5.0 ± 2.0	3.9 ± 1.8	4.6 ± 1.9	5.9 ± 2.1	4.3 ± 1.9	5.2 ± 2.0	4.8 ± 2.0	4.3 ± 1.9	6.1 ± 2.2	5.9 ± 2.1
GPQA	Qwen2.5-3B	17.4 ± 3.5	16.1 ± 3.4	16.7 ± 3.5	16.3 ± 3.4	19.6 ± 3.7	20.8 ± 3.8	18.1 ± 3.6	18.1 ± 3.6	17.2 ± 3.5	14.5 ± 3.3	17.2 ± 3.5	18.8 ± 3.6	19.4 ± 3.7
	Qwen2.5-7B	16.7 ± 3.5	19.4 ± 3.7	17.2 ± 3.5	15.6 ± 3.4	20.3 ± 3.7	17.9 ± 3.5	21.4 ± 3.8	18.3 ± 3.6	19.6 ± 3.7	16.7 ± 3.5	18.5 ± 3.6	20.1 ± 3.7	19.4 ± 3.7
	Qwen2.5-14B	15.4 ± 3.3	21.2 ± 3.8	21.0 ± 3.8	21.0 ± 3.8	21.4 ± 3.8	24.1 ± 4.0	24.6 ± 4.0	19.9 ± 3.7	20.3 ± 3.7	19.2 ± 3.6	26.8 ± 4.1	24.6 ± 4.0	23.0 ± 3.9
	Llama-3.2-3B	14.7 ± 3.3	15.4 ± 3.3	15.6 ± 3.4	14.3 ± 3.2	16.1 ± 3.4	15.4 ± 3.3	15.2 ± 3.3	14.3 ± 3.2	14.5 ± 3.3	15.0 ± 3.3	15.8 ± 3.4	15.2 ± 3.3	14.1 ± 3.2
	Llama-3.1-8B	14.3 ± 3.2	10.5 ± 2.8	10.5 ± 2.8	12.5 ± 3.1	14.5 ± 3.3	13.8 ± 3.2	10.7 ± 2.9	12.3 ± 3.0	12.1 ± 3.0	9.4 ± 2.7	13.4 ± 3.2	10.5 ± 2.8	11.4 ± 2.9
	Mistral-7B	9.6 ± 2.7	8.0 ± 2.5	6.7 ± 2.3	4.0 ± 1.8	0.9 ± 0.9	1.6 ± 1.1	4.5 ± 1.9	5.1 ± 2.0	3.3 ± 1.7	4.7 ± 2.0	4.0 ± 1.8	2.9 ± 1.6	3.8 ± 1.8
GSM8K	Qwen2.5-3B	81.9 ± 5.3	88.0 ± 4.5	89.0 ± 4.3	89.0 ± 4.3	88.0 ± 4.5	89.0 ± 4.3	88.5 ± 4.4	87.5 ± 4.6	86.5 ± 4.7	86.0 ± 4.8	87.5 ± 4.6	88.5 ± 4.4	88.0 ± 4.5
	Qwen2.5-7B	85.3 ± 4.9	89.0 ± 4.3	86.0 ± 4.8	91.5 ± 3.9	92.0 ± 3.8	94.0 ± 3.3	93.5 ± 3.4	86.5 ± 4.7	90.5 ± 4.1	92.5 ± 3.7	92.5 ± 3.7	94.0 ± 3.3	92.5 ± 3.7
	Qwen2.5-14B	90.4 ± 4.1	94.5 ± 3.2	93.0 ± 3.5	94.5 ± 3.2	95.0 ± 3.0	95.0 ± 3.0	96.0 ± 2.7	95.0 ± 3.0	94.5 ± 3.2	95.5 ± 2.9	94.5 ± 3.2	96.5 ± 2.5	95.0 ± 3.0
	Llama-3.2-3B	62.3 ± 6.7	69.5 ± 6.4	67.0 ± 6.5	71.5 ± 6.3	66.0 ± 6.6	69.5 ± 6.4	68.0 ± 6.5	67.5 ± 6.5	65.0 ± 6.6	67.0 ± 6.5	65.0 ± 6.6	66.5 ± 6.5	68.0 ± 6.5
	Llama-3.1-8B	78.0 ± 5.7	83.0 ± 5.2	84.5 ± 5.0	84.0 ± 5.1	85.5 ± 4.9	82.5 ± 5.3	83.0 ± 5.2	85.0 ± 4.9	81.0 ± 5.4	80.5 ± 5.5	84.0 ± 5.1	82.5 ± 5.3	84.5 ± 5.0
	Mistral-7B	5.0 ± 3.0	32.0 ± 6.5	12.0 ± 4.5	10.5 ± 4.2	4.0 ± 2.7	38.5 ± 6.7	21.0 ± 5.6	4.0 ± 2.7	39.5 ± 6.8	3.5 ± 2.5	39.5 ± 6.8	36.0 ± 6.7	33.0 ± 6.5
HLE	Qwen2.5-3B	2.0 ± 0.6	2.4 ± 0.6	2.1 ± 0.6	1.9 ± 0.5	2.5 ± 0.6	2.4 ± 0.6	2.2 ± 0.6	1.9 ± 0.6	2.4 ± 0.6	2.3 ± 0.6	2.3 ± 0.6	2.2 ± 0.6	2.5 ± 0.6
	Qwen2.5-7B	2.0 ± 0.6	2.4 ± 0.6	1.9 ± 0.6	2.1 ± 0.6	2.5 ± 0.6	2.2 ± 0.6	2.6 ± 0.6	1.9 ± 0.6	2.6 ± 0.6	2.2 ± 0.6	2.5 ± 0.6	2.1 ± 0.6	2.3 ± 0.6
	Qwen2.5-14B	1.9 ± 0.5	2.1 ± 0.6	2.4 ± 0.6	2.7 ± 0.6	2.2 ± 0.6	1.7 ± 0.5	2.5 ± 0.6	2.4 ± 0.6	1.9 ± 0.6	2.4 ± 0.6	2.7 ± 0.6	1.9 ± 0.5	2.8 ± 0.7
	Llama-3.2-3B	0.3 ± 0.2	1.5 ± 0.5	1.3 ± 0.5	1.4 ± 0.5	1.1 ± 0.4	1.1 ± 0.4	1.2 ± 0.4	1.1 ± 0.4	0.8 ± 0.4	1.1 ± 0.4	1.1 ± 0.4	1.8 ± 0.5	1.4 ± 0.5
	Llama-3.1-8B	2.2 ± 0.6	2.6 ± 0.6	2.7 ± 0.6	2.5 ± 0.6	2.2 ± 0.6	2.5 ± 0.6	2.1 ± 0.6	2.2 ± 0.6	2.4 ± 0.6	2.1 ± 0.6	2.1 ± 0.6	2.3 ± 0.6	2.8 ± 0.7
	Mistral-7B	0.3 ± 0.2	1.2 ± 0.4	1.1 ± 0.4	0.9 ± 0.4	0.3 ± 0.2	0.8 ± 0.4	0.9 ± 0.4	0.6 ± 0.3	0.9 ± 0.4	0.6 ± 0.3	0.6 ± 0.3	0.5 ± 0.3	1.0 ± 0.4
Math401	Qwen2.5-3B	57.4 ± 4.8	58.6 ± 4.8	57.4 ± 4.8	60.3 ± 4.8	60.1 ± 4.8	60.1 ± 4.8	58.6 ± 4.8	58.4 ± 4.8	58.4 ± 4.8	58.1 ± 4.8	58.1 ± 4.8	59.4 ± 4.8	58.6 ± 4.8
	Qwen2.5-7B	57.9 ± 4.8	58.9 ± 4.8	57.4 ± 4.8	61.6 ± 4.8	61.8 ± 4.8	62.3 ± 4.7	60.1 ± 4.8	61.3 ± 4.8	58.6 ± 4.8	61.1 ± 4.8	60.1 ± 4.8	64.1 ± 4.7	62.1 ± 4.7
	Qwen2.5-14B	64.1 ± 4.7	67.3 ± 4.6	67.8 ± 4.6	67.1 ± 4.6	66.3 ± 4.6	67.8 ± 4.6	67.6 ± 4.6	67.1 ± 4.6	66.8 ± 4.6	67.1 ± 4.6	66.8 ± 4.6	67.8 ± 4.6	68.3 ± 4.6
	Llama-3.2-3B	45.9 ± 4.9	49.9 ± 4.9	50.1 ± 4.9	49.4 ± 4.9	51.4 ± 4.9	48.6 ± 4.9	50.6 ± 4.9	50.4 ± 4.9	48.1 ± 4.9	52.9 ± 4.9	51.1 ± 4.9	47.1 ± 4.9	50.6 ± 4.9
	Llama-3.1-8B	50.6 ± 4.9	52.6 ± 4.9	52.6 ± 4.9	50.6 ± 4.9	50.4 ± 4.9	48.6 ± 4.9	51.4 ± 4.9	50.9 ± 4.9	51.4 ± 4.9	50.6 ± 4.9	47.9 ± 4.9	48.6 ± 4.9	51.9 ± 4.9
	Mistral-7B	0.0 ± 0.1	11.7 ± 3.1	9.2 ± 2.8	5.2 ± 2.2	7.0 ± 2.5	7.0 ± 2.5	1.7 ± 1.3	28.4 ± 4.4	8.7 ± 2.8	0.5 ± 0.7	14.7 ± 3.5	8.7 ± 2.8	3.7 ± 1.9
Math500	Qwen2.5-3B	52.8 ± 4.4	49.8 ± 4.4	50.6 ± 4.4	51.8 ± 4.4	51.8 ± 4.4	51.6 ± 4.4	51.2 ± 4.4	52.4 ± 4.4	51.6 ± 4.4	52.0 ± 4.4	51.8 ± 4.4	52.2 ± 4.4	51.0 ± 4.4
	Qwen2.5-7B	46.8 ± 4.4	54.0 ± 4.4	57.6 ± 4.3	59.0 ± 4.3	60.2 ± 4.3	60.4 ± 4.3	60.0 ± 4.3	55.0 ± 4.4	56.8 ± 4.3	61.0 ± 4.3	61.4 ± 4.3	60.6 ± 4.3	60.2 ± 4.3
	Qwen2.5-14B	61.6 ± 4.3	63.2 ± 4.2	62.0 ± 4.3	62.8 ± 4.2	62.2 ± 4.3	64.4 ± 4.2	63.4 ± 4.2	60.6 ± 4.3	63.0 ± 4.2	62.8 ± 4.2	64.4 ± 4.2	63.2 ± 4.2	62.6 ± 4.2
	Llama-3.2-3B	26.6 ± 3.9	32.4 ± 4.1	30.2 ± 4.0	32.6 ± 4.1	34.4 ± 4.2	34.8 ± 4.2	31.8 ± 4.1	31.2 ± 4.1	32.6 ± 4.1	31.0 ± 4.1	34.8 ± 4.2	33.2 ± 4.1	32.6 ± 4.1
	Llama-3.1-8B	33.8 ± 4.1	38.2 ± 4.3	35.6 ± 4.2	35.8 ± 4.2	36.6 ± 4.2	33.0 ± 4.1	36.2 ± 4.2	34.6 ± 4.2	35.6 ± 4.2	36.4 ± 4.2	36.6 ± 4.2	36.2 ± 4.2	35.8 ± 4.2
	Mistral-7B	2.2 ± 1.3	9.8 ± 2.6	8.4 ± 2.4	6.8 ± 2.2	6.0 ± 2.1	10.0 ± 2.6	8.4 ± 2.4	4.0 ± 1.7	11.2 ± 2.8	4.4 ± 1.8	12.6 ± 2.9	10.8 ± 2.7	10.8 ± 2.7

Table 4: ***Targeted complexity buckets can outperform larger mixed-data training.*** Accuracy (%) after fine-tuning on the full INSTRUCT CC dataset (∼40k samples) compared with the best-performing individual complexity bucket (∼8k samples). Δ reports Full − Best, so negative values indicate that the targeted bucket outperforms full-dataset training. The split of the best-performing bucket is shown in parentheses.

	`Qwen2.5-7B`			`Llama-3.2-3B`		
Benchmark	**Full dataset (%)**	**Best bucket (%)**	**Δ** (%)	**Full (%)**	**Best bucket (%)**	**Δ** (%)
GSM8K	83.47	94.00 (MAX)	−10.53	67.93	71.50 (MID)	−3.57
MATH401	60.85	62.34 (MAX)	−1.50	50.37	51.37 (HIGH)	−1.00
MATH500	47.20	60.40 (MAX)	−13.20	31.80	34.80 (MAX)	−3.00
GPQA	16.29	20.31 (HIGH)	−4.02	14.96	16.07 (HIGH)	−1.12
BBEH-MINI	7.61	10.65 (MIN)	−3.04	2.39	3.26 (HIGH)	−0.87
HLE	1.69	2.53 (HIGH)	−0.84	1.05	1.48 (MIN)	−0.42
Average	36.18	41.07 (MAX)	−4.89	28.08	28.74 (MAX)	−0.66

We emphasise that this experiment is not intended to maximise performance or exhaustively study scaling behaviour. Rather, it serves as a controlled check on whether the observed complexity effects disappear once more data is added. The results suggest that they do not, supporting the practical relevance of complexity-aware data selection, while leaving larger-scale validation across more models, datasets, and training stages as an important direction for future work.

G Correlation Calculation Details

To rigorously quantify the relationship between training code data complexity and model reasoning performance, we employ Spearman's rank correlation coefficient (ρ) (Dodge, 2008). This non-parametric measure is chosen because our complexity levels (MIN, LOW, MID, HIGH, MAX). represent an ordinal scale rather than a continuous ratio scale, making Pearson's correlation less appropriate. Specifically, for each base model and dataset combination, we calculate ρ between the integer-mapped complexity levels $(0 - 4)$ and the corresponding evaluation accuracy. The calculation is performed using the `scipy.stats.spearmanr` function from the `SciPy`[10] library, which also provides two-sided p-values to assess statistical significance. A standard significance level of $\alpha = 0.05$ is used to determine if the observed correlations are statistically significant. We exclude control groups from this specific analysis to focus solely on the trend within the complexity-stratified fine-tuning runs.

[10]SciPy: https://scipy.org/