Quantifying Error Tolerance in Synthetic Data: An Atomic-level Operand vs. Operator Perturbation Study

Anonymous ACL submission

Abstract

Synthetic data generation has become a cornerstone for advancing large language models. However, the absence of the quantitative analysis for error tolerance became a critical bottleneck. Consequently, current filtering strategies fluctuate between two extremes: they are either overly aggressive, risking the exclusion of potentially valuable samples, or overly permissive, failing to eliminate erroneous samples effectively. To bridge this gap, this paper introduces **A**tomic **T**ree **O**peration **M**odeling (**ATOM**), a framework that decomposes data into functional units ($f(x) \rightarrow y$). ATOM distinguishes *benign Operand x perturbations* from *fatal Operator f perturbations*. The former are needlessly discarded by aggressive filtering, while the latter slip through permissive filtering. Our experiments reveal a **double dissociation**: models are robust to operand perturbations but collapse under operator perturbations. By prioritizing operator over aggressive operand precision, our ATOM-synthesized data significantly outperforms rigorous baselines (e.g., +3.1% gain over LIMA), suggesting that operator diversity matters more than operand precision.

1 Introduction

The ability of Large Language Models (LLMs) heavily relies on training data (Grattafiori et al., 2024; OpenAI et al., 2024; Qwen-Team, 2024; DeepSeek-AI et al., 2025; Yang et al., 2025a; Gemini-Team et al., 2025). As high-quality human-generated data becomes increasingly scarce, the field has largely adopted synthetic data generation (Taori et al., 2023; Zhou et al., 2023; Wang et al., 2023b; Xu et al., 2024, 2025; Maosong et al., 2025) to distill capabilities from stronger models. However, a major bottleneck remains: teacher models inevitably hallucinate and introduce factual errors.

Studies such as LIMA (Zhou et al., 2023) underscored that data quality is a decisive factor in supervised fine-tuning (SFT). However, they lack a framework to characterize which errors actually harm SFT. Lacking such an error tolerance criterion (Frenay and Verleysen, 2014), current filtering methodologies are forced into two extremes: the first is *aggressive filtering* (Zhou et al., 2023; Xu et al., 2024; Maosong et al., 2025), which risks discarding valuable samples. The second is *permissive filtering* (Li et al., 2024; Zhao et al., 2024; Li et al., 2025), which retains harmful ones. Unfortunately, existing quantifying error tolerance methods still fall short: they are limited to either coarse, qualitative assessments that neglect error quantification (Cho, 2024a; Raghavendra et al., 2025), or linguistic surface taxonomies that overlook essential logical properties (Alajrami et al., 2025). **More importantly, principled filtering hinges on quantifying which errors a model can tolerate and to what extent, which remains undefined.**

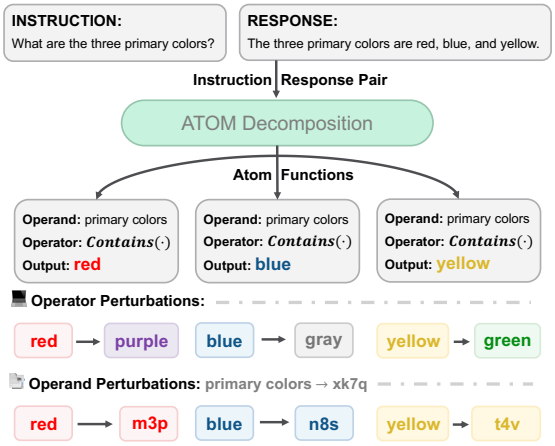


Figure 1: The illustration of extract atomic functions.

To rigorously identify the error tolerance in synthetic data, this paper proposes an **A**tomic **T**ree **O**peration **M**odeling (**ATOM**) framework, grounded in Problem Space Theory, which models problem-solving as applying operators to transform inputs into outputs (Newell, 1972). As shown in Figure 1, ATOM decomposes instruction-response pairs into atomic operations: $f(x) \rightarrow y$. For

example, the response "The three primary colors are red, blue, and yellow." is decomposed as $Contains$(primary colors) $\rightarrow$ red/blue/yellow. ATOM makes error tolerance quantifiable by turning each response into countable operation units, where we can precisely control the error type and perturbation ratio. Specifically, based on whether the logical operator f is preserved, ATOM defines two error types: *Operand Perturbations* modify x while updating y to keep f valid, e.g., $Contains$(xk7q) $\rightarrow$ m3p/n8s/t4v. And *Operator Perturbations* keep x fixed but modify y to violate f, e.g., $Contains$(primary colors) $\rightarrow$ purple/gray/green. This taxonomy allows us to investigate and compare the model's tolerance to *operand and operator perturbations*.

Specifically, this paper constructs an ATOM operator tree to generate Atomic-QA. Explicit atomic operations of Atomic-QA enable controlled injection of *operand and operator perturbations*. Experiments on these two variants reveal a clear asymmetry: models are highly tolerant to *operand perturbations*, with downstream performance remaining stable even when operands are replaced by random strings and yielding only a **2.0%** relative decrease. Centered Kernel Alignment analysis (Kornblith et al., 2019) further shows that the resulting internal representations remain aligned with those learned from the original Atomic-QA. In contrast, *operator perturbations* substantially degrade performance, causing a **18.2%** relative decrease. These findings support an operator-centered principle: useful synthetic data should preserve valid and diverse operators rather than strict operand precision.

Guided by the above operator-centered principle, this paper further validate the effectiveness of synthetic data with Atomic-QA and Atomic-LIFD subsets that preserve broad operator coverage from the ATOM operator tree, retaining structural diversity often discarded by *aggressive filtering*. Across four model backbones and 10 benchmark tasks, these ATOM-guided datasets consistently outperform standard baselines, including Alpaca (Taori et al., 2023), LIMA (Zhou et al., 2023), WizardLM (Xu et al., 2025) and Magpie(Xu et al., 2024).

We also apply the generalizable Atomic-X pipeline to existing instruction and math datasets, producting Atomic-Alpaca and Atomic-MetaMath with the controlled perturbations. The results further confirm our findings and prove the generalization of the ATOM. Our contributions are as follows:

- **ATOM Framework:** We construct **ATOM**, a unified framework for defining, quantifying, and controlling error tolerance in synthetic data.
- **Operator–Operand Asymmetry:** We discover that models are sensitive to *operator perturbations* but robust to *operand perturbations*.
- **Operator-Centered Synthesis:** Guided by this principle, we construct Atomic-QA, which achieves strong performance. And this paper further generalize the error tolerance findings to existing datasets via the Atomic-X pipeline.

2 Related Work

Construction and Filtering of Synthetic Data Prior research establishes that SFT prioritizes data quality over sheer scale, with works like LIMA (Zhou et al., 2023) and Textbooks Are All You Need (Gunasekar et al., 2023) proving that effective alignment can emerge from small, high-quality corpora. Consequently, pipelines such as Self-Instruct (Wang et al., 2023b) and WizardLM (Xu et al., 2025) employ generate-then-filter strategies to synthesize data, while other approaches focus on selection via heuristic-based strategies (Zhao et al., 2024; Li et al., 2024, 2025) or contribution estimation (Dai et al., 2025; Chen et al., 2025b; Jiang et al., 2025). However, current methods lack a quantitative framework for defining data quality, leading to a trade-off between overly aggressive filtering and overly permissive retention.

Learning Mechanisms in Noisy Data Prior work on noisy SFT investigates various settings. FTNI (Alajrami et al., 2025) found that perturbed instructions can surprisingly improve performance. TInt (Havrilla and Iyer, 2024) and FACO (Cho, 2024b) explored model robustness to erroneous reasoning chains. Recent evidence challenges the superficial alignment hypothesis (Zhou et al., 2023), showing that SFT goes beyond style adaptation to genuinely enhance reasoning and follow pre-training scaling laws (Chen et al., 2025a; Raghavendra et al., 2025). However, current research lacks fine-grained analysis, failing to determine noise thresholds. Details are listed in Appendix A.

3 Methodology

This section details the **A**tomic **T**ree **O**peration **M**odeling (**ATOM**) framework. As depicted in Figure 2, the proposed framework operates in two distinct phases: 1) Atomic Tree Construction (§3.1) and 2) Atomic Data Synthesis (§3.2).

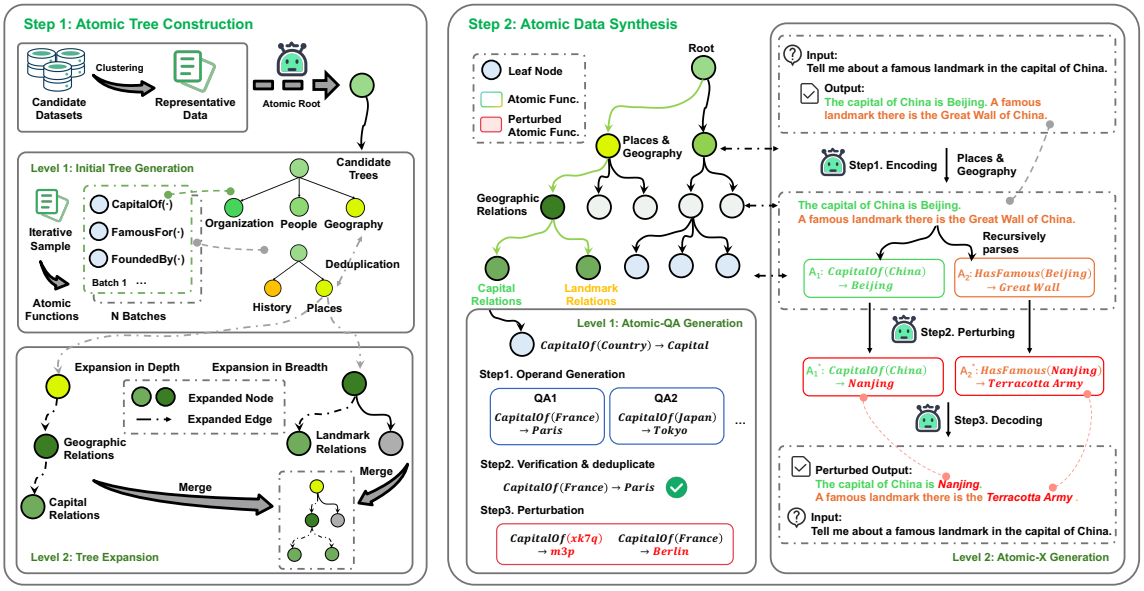


Figure 2: The pipeline of our **A**tomic **T**ree **O**peration **M**odeling (**ATOM**) framework. ATOM consists of two phases: 1) Atomic Tree Construction, which builds an operator hierarchy where leaf nodes define concrete atomic operators, and 2) Atomic Data Synthesis, which supports both Atomic-QA and Atomic-X.

3.1 Atomic Tree Construction

To formalize the construction process, we first introduce the concept of the Atomic Function.

3.1.1 Atomic Function

Definition 1 (Atomic Function)**.** An Atomic Function is defined as the minimal, logically indivisible unit of execution, denoted as a triplet $\mathcal{A} = \langle x, f, y \rangle$, where we have $f(x) \rightarrow y$.

Formally, it represents a deterministic mapping:

- f (**The Operator**): The abstract logical rule or relational predicate that governs the transformation. Operators are reasoning patterns that define what transformation to apply, e.g., $CapitalOf(\cdot)$, $FamousFor(\cdot)$, or $FoundedBy(\cdot)$ in Figure 2.
- x (**The Operand**): The concrete, substitutable input value upon which the operator acts. Operands are domain-specific factual content: they specify which particular instance the operator processes, and can be replaced without altering the underlying logic, e.g., *Paris*, 100, *primary colors*.
- y (**The Output**): The deterministic result of applying f to x: $CapitalOf$(France) $\rightarrow$ Paris.

An Atomic Function is defined such that any further decomposition of the operator f would result in a loss of semantic intent or invalidate the executable logic within the task context. As shown in Figure 1, the instruction *List the primary colors* with response *Red, blue, and yellow* is decomposed into atomic function: $Contains$(primary colors) $\rightarrow$ {red, blue, yellow}, where $Contains$ is the **operator** and primary colors is the **operand**. An *operand perturbation* jointly modifies (x, y) so that $f(x') = y'$ still holds (e.g., xk7q) while preserving the operator, yielding $Contains$(xk7q) $\rightarrow$ {m3p/n8s/t4v}. This result is factually different but logically coherent. In contrast, an *operator perturbation* would modify y alone so that $f(x) \neq y'$, for example $Contains$(primary colors) $\rightarrow$ {purple, orange, green}, producing an incoherent result that violates the operator's logic. **To further clarify the atomic function, we provide more examples in Appendix G.**

3.1.2 Pipeline of Atomic Tree Construction

We organize atomic operators into a tree-structured hierarchy. For instance, the path from tree: `Root → Geography & Places → Geographic Relations → Capital Relations` reaches a leaf operator $CapitalOf(\cdot)$. This hierarchical structure naturally aligns with the recursive composition of complex tasks: a multi-step reasoning problem decomposes through intermediate categories until reaching semantically irreducible leaf operators. Consequently, our goal is to construct a complete operator hierarchy whose leaf nodes cover the space of atomic operators required by a given domain.

As shown in Figure 2, Step 1 starts with Representative Data Selection. To mitigate computational overhead while ensuring coverage, we encode candidate seed datasets using bge-m3 (Chen et al., 2024). We then apply KCenterGreedy (Sener and Savarese, 2018) clustering to select samples

that maximize the coverage of the semantic space and ensure the extracted subset diversity.

After representative data selection, **Level 1: Initial Tree Generation** employs an iterative strategy in which o3-mini processes batches of atomic functions sampled from representative data and assigns each function to a category. Within each batch, the categories are organized into candidate sub-trees, where upper nodes represent coarser domains and finer nodes represent more specific operator categories. For example, atomic functions such as $CapitalOf$(France) $\rightarrow$ Paris are grouped under the category path `Root` $\rightarrow$ `Geography & Places` $\rightarrow$ $CapitalOf(\cdot)$, where the leaf category anchors the concrete operator. As shown in the left-middle of Figure 2, the candidate sub-trees produced from successive batches are iteratively merged into a global tree. During each merge, an LLM-based semantic assessment module identifies category nodes in the incoming sub-tree whose semantics overlap with existing ones (e.g., `Places & Geography` v.s. `Geography & Places`). **The initial trees are ultimately verified by 4 human experts' votes, detailed in Appendix B.1.**

After initialization, in **Level 2: Tree Expansion**, the generated tree is expanded with the strict atomic constraints defined in our definition. This phase iteratively refines category nodes through a dual-path expansion strategy inspired by EpiCoder (Wang et al., 2025): Expansion in Depth for granularity and Expansion in Breadth for coverage. Following the example above, the initial tree contains a coarse category path `Root` $\rightarrow$ `Geography & Places`. *Expansion in Depth* recursively subdivides category nodes by inserting finer-grained relation classes, yielding $\cdots \rightarrow$ `Geographic Relations` $\rightarrow$ `Capital Relations`, progressively narrowing the scope until each leaf is irreducible as judged by the LLM or capped at a maximum depth. *Expansion in Breadth* supplements sibling category nodes at the same level, e.g., adding sibling nodes like `Landmark Relations` alongside `Capital Relations`. As shown in the left-bottom panel of Figure 2, the resulting expanded nodes are consolidated back into the main hierarchy through a final merge operation. **Detailed examples and prompts are shown in Appendix B.1. Leaf nodes are sampled and verified by expert voting.**

3.2 Atomic-QA Synthesis

After obtaining the atomic tree, we synthesize training data and its controlled perturbation variants through the following pipeline. Atomic-QA Generation synthesizes training data starting from the leaf nodes of the atomic tree. Each leaf node defines a concrete atomic operator f like $CapitalOf(\cdot)$, and the generation proceeds through:

Step 1 (QA Generation): Given a leaf node and its associated operator f, we first generate diverse operands x in its domain, and then compute the corresponding output $y = f(x)$ to form atomic triplets $\mathcal{A} = \langle x, f, y \rangle$. For example, for $f = CapitalOf$, we generate operands such as France, Japan, Brazil and compute their outputs Paris, Tokyo, Brasília.

Step 2 (Verification): To ensure data quality and diversity, we deduplicate overlapping operands x across neighboring nodes and rigorously verify the logical validity of the generated triplets, producing the **Atomic-QA** corpus.

Step 3 (Perturbation): We inject noise to create variants. Specifically, *Operand perturbation* replaces x with a random string and adjusts y while preserving f, e.g., $CapitalOf$(xk7q) $\rightarrow$ m3p, giving structurally coherent but factually meaningless samples. *Operator perturbation* alters y while keeping x fixed, such that the resulting pair (x, y') encodes a mapping that contradicts f, e.g., $CapitalOf$(France) $\rightarrow$ Berlin. At last, two variants, *Operand-Perturbed Atomic-QA* and *Operator-Perturbed Atomic-QA* are generated.

4 Experimental Setup

In the following sections, we exploit experiments to answer the following research questions (**RQs**):

- **RQ1:** Which plays the dominant role in training: the **Operand** (x) or the **Operator** (f)? (§5)
- **RQ2:** Does ATOM-synthesized data outperform aggressively-filtered baselines? (§6)
- **RQ3:** Whether the error tolerance extends to the real synthesis data setting. (§7)

Dataset: To address RQ1 and RQ2, we construct an operator tree following Section 3.2 using FB15k (Schlichtkrull et al., 2018), ZsRE (Tran et al., 2022), and wiki_recent (Wang et al., 2023a) as seeds, and synthesize the Atomic-QA dataset (1.37M instances) along with its operand-perturbed and operator-perturbed variants. For RQ3, we construct Atomic-Alpaca and Atomic-MetaMath via the Atomic-X pipeline (§7), injecting operator errors at controlled rates into real-world complex data. See Appendix C.1 for details.

Model: To ensure generalizability, we evaluate LLaMA3.1 (8B) (Grattafiori et al., 2024), Qwen2.5

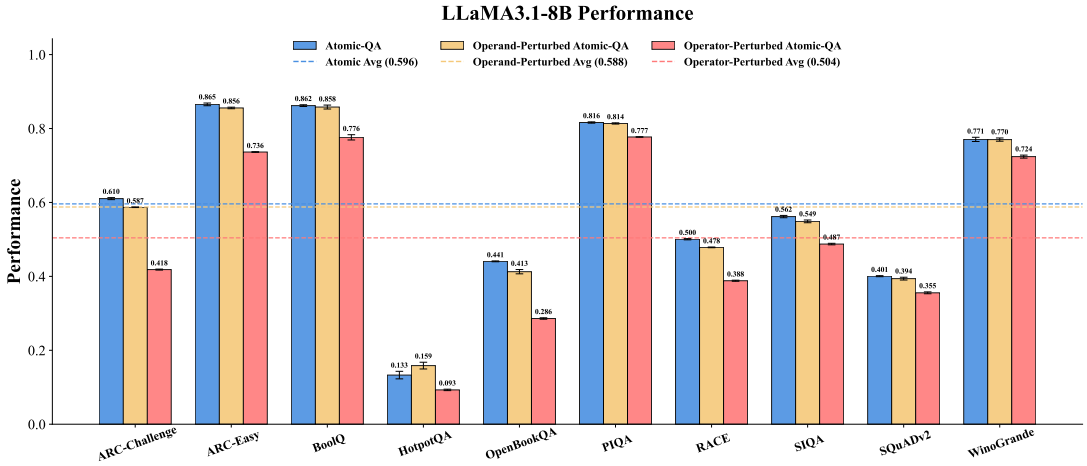


Figure 3: Performance comparison of LLaMA3.1 across Atomic-QA, Operand-Perturbed Atomic-QA, and Operator-Perturbed Atomic-QA. Dashed lines denote the average performance for each setting.

(7B & 14B) (Qwen-Team, 2024), and Qwen3 (8B) (Yang et al., 2025a). Training is conducted via Llama-Factory (Zheng et al., 2024b) and evaluation via the LM Evaluation Harness (Gao et al., 2024). Configuration details are in Appendix C.2.

5 Operator–Operand Asymmetry

To evaluate operator–operand tolerance during training, we construct two controlled variants of Atomic-QA using the perturbation strategies defined in §3.2. *Operand-Perturbed Atomic-QA* applies operand perturbation: each operand x is replaced with a random string of the same character length while preserving the operator f, producing data that is factually meaningless but structurally coherent. *Operator-Perturbed Atomic-QA* applies operator perturbation: following Xie et al. (2024), we use an LLM to rewrite the answers and verify hallucinations, thereby corrupting the operator logic f while keeping operands intact. This data is superficially fluent but logically incoherent.

5.1 Performance Comparison

Figure 3 presents the comparative results on LLaMA3.1-8B, revealing a double dissociation: models are robust to operand corruption but collapse under operator corruption. **Results of other models are listed in Figure 14-16 of Appendix.**

Findings 1: Robustness to Operand Noise: The resilience to operand noise is not limited to simple tasks but generalizes consistently across downstream evaluations of varying complexity. As shown in Figure 3, the performance of Operand-Perturbed Atomic-QA closely tracks the baseline Atomic-QA across all datasets. On reasoning-heavy tasks like ARC-Challenge, the performance gap is minimal (0.587 v.s. 0.610), and on the complex multi-hop reasoning task HotpotQA, the model trained on random strings even slightly outperforms the baseline (0.159 v.s. 0.133). This demonstrates that even in complex scenarios, the model's performance relies primarily on acquiring the operator logic f rather than memorizing the specific atomic operands x. The results shown in Figures 14, 15, and 16 indicate that our findings generalize across different model structures.

Findings 2: Sensitivity to Operator Noise: In stark contrast, when trained on Operator-Perturbed Atomic-QA, where the structural logic is disrupted (corrupting the operator f), performance plummets to **50.4%**. This represents a substantial absolute decline of **9.2%** (a relative drop of ∼18.2%), which highlights the model's acute sensitivity to the operation perturbation which may corrupt the structural integrity in the training data.

5.2 Mechanism Validation

One might concern that Operand-Perturbed Atomic-QA's strong performance comes from metric hacking, surface coherence (Appendix D.4), or superficial heuristics rather than structural integrity. To rule this out, we analyze the training dynamics and latent representations, showing that the model learns consistently with the unperturbed setting.

Behavioral Alignment: We constructed a held-out diagnostic test set by sampling subsets from Atomic-QA and rewriting them into multiple-choice questions (MCQs). We then monitored the validation metrics throughout the training process. As shown in Figure 4 (Left), the accuracy curves for

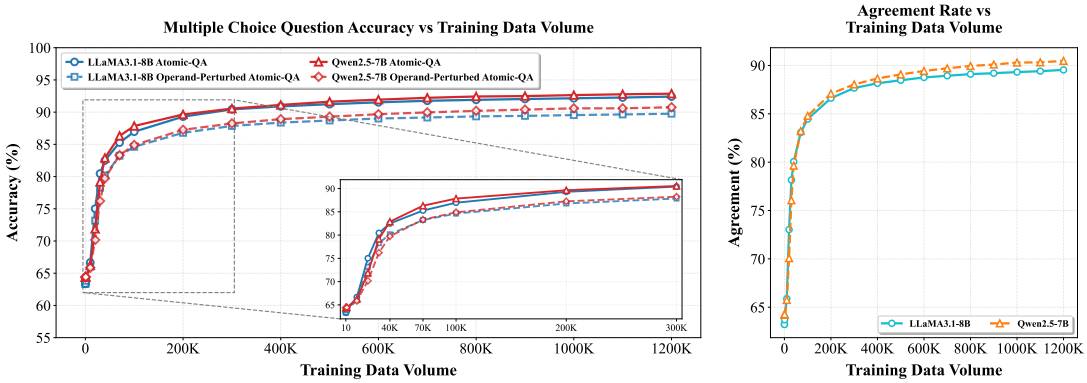


Figure 4: MCQ accuracy vs. training data scale (main plot), with model agreement under replaced vs. original entity settings. Agreement Rate is the proportion of questions both models answer correctly.

models trained on both Atomic-QA and Operand-Perturbed Atomic-QA rise in unison, converging to nearly identical performance levels. Complementing this, Figure 4 (Right) illustrates the Agreement Rate—the proportion of test instances where both models predict correctly. The steady increase in agreement demonstrates that as training progresses, the models do not just achieve similar scores. They converge on the same behavioral patterns. This synchrony indicates that the optimization landscape is dominated by the structural operator f, rendering atomic operands (x) irrelevant to learning.

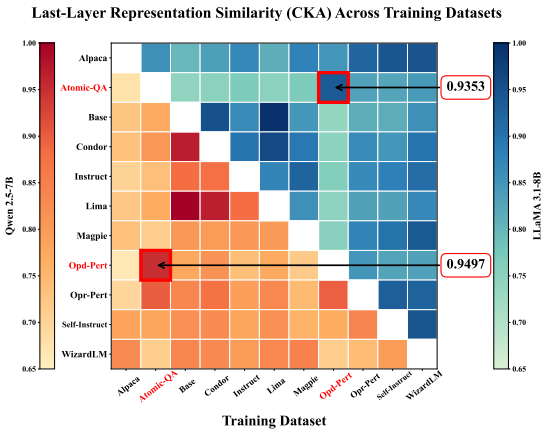


Figure 5: CKA similarity of the output hidden states between different trained models. The lower part represents the CKA similarity of Qwen 2.5-7B, while the upper part represents LLaMA 3.1-8B.

Representational Alignment: Do these models merely behave similarly, or do they actually learn the same internal representations? We utilize Centered Kernel Alignment (CKA) (Kornblith et al., 2019) to measure the similarity of the last-layer hidden states between different models. Figure 5 presents the CKA heatmap. Strikingly, the model trained on Operand-Perturbed Atomic-QA exhibits an exceptionally high similarity score (e.g., **0.9497** on Qwen2.5-7B) with the model trained on the standard *Atomic-QA*. This alignment suggests that the models have learned equivalent internal representations, despite their differing training data. Training dynamics of PPL and CKA are in Appendix D (Figures 12, 13). Together, these findings reveal an asymmetry and motivate an operator-centered principle: **synthetic data should prioritize operator coverage over strict operand filtering.**

6 Effectiveness of Atomic-QA

We validate the operator-centered principle through extensive comparisons against existing baselines.

6.1 Atomic-QA Outperforms Baselines

Tables 1 and 3 report evaluation results across 10 benchmarks and four backbones.

Surpassing Aggressive Filtering Baselines: We verify the operator-centered principle by comparing Atomic-QA against LIMA, a canonical aggressive-filtering baseline. On LLaMA3.1-8B, Atomic-QA achieves an absolute gain of **+3.1%** over LIMA (56.5% average). This trend persists across model scales, with Atomic-QA outperforming LIMA by 1.7% on Qwen2.5-7B and 2.5% on Qwen3-8B. This gap underscores that aggressively filtering methods discard valuable training signals, capping the model's potential. To further isolate the effect of operator coverage, we also evaluate Atomic-LIFD, a compact subset of Atomic-QA selected to retain diverse operator structures. **Despite using minimal training tokens (Appendix D.1), Atomic-LIFD achieves competitive results. This confirms that operator-level structural diversity is the driver of performance gains.**

Superiority over Standard Synthetic Baselines: Compared to widely used synthetic datasets

Table 1: Overall performance across all training datasets on LLaMA3.1-8B and Qwen3-8B. Color-coded cells (green/red) are used to indicate performance above or below the mean across methods (All Average), with darker colors representing better or worse performance. **The results for Qwen2.5-7B and Qwen2.5-14B are in Table 3.**

Model Family		ARC-Challenge	ARC-Easy	BoolQ	Hotpot-QA	Open-BookQA	PIQA	RACE	SIQA	SQuADv2	Wino-Grande	Avg
LLaMA3.1-8B	Base	0.549	0.847	0.820	0.106	0.366	0.807	0.433	0.529	0.375	0.765	0.560
	All Average	0.564	0.850	0.855	0.127	0.387	0.812	0.464	0.548	0.381	0.768	0.576
	Instruct	0.558	0.854	0.870	0.209	0.362	0.811	0.464	0.548	0.420	0.764	0.586
	LIMA	0.552	0.848	0.828	0.134	0.368	0.808	0.432	0.529	0.377	0.773	0.565
	Condor	0.563	0.854	0.855	0.117	0.382	0.811	0.470	0.546	0.378	0.770	0.575
	Magpie	0.575	0.845	0.859	0.122	0.380	0.808	0.464	0.563	0.374	0.761	0.575
	WizardLM	0.580	0.862	0.858	0.107	0.396	0.823	0.478	0.573	0.385	0.766	0.583
	Alpaca	0.562	0.846	0.867	0.115	0.406	0.819	0.469	0.576	0.393	0.776	0.583
	Self-Instruct	0.469	0.811	0.839	0.097	0.352	0.803	0.430	0.511	0.321	0.739	0.537
	Wiki	0.562	0.856	0.845	0.132	0.370	0.810	0.453	0.530	0.386	0.778	0.572
	Wiki-Rewrite	0.560	0.855	0.847	0.100	0.364	0.811	0.456	0.528	0.388	0.777	0.569
	Atomic-LIFD	0.607	0.858	0.875	0.133	0.440	0.812	0.494	0.568	0.364	0.770	0.592
	Atomic-QA	0.610	0.865	0.862	0.133	0.441	0.816	0.500	0.562	0.401	0.771	0.596
Qwen3-8B	Base	0.631	0.871	0.871	0.233	0.370	0.799	0.473	0.566	0.444	0.743	0.600
	All Average	0.631	0.870	0.875	0.183	0.375	0.798	0.489	0.562	0.420	0.746	0.595
	Instruct	0.629	0.873	0.871	0.114	0.378	0.782	0.491	0.567	0.430	0.707	0.584
	LIMA	0.632	0.870	0.873	0.237	0.374	0.803	0.478	0.558	0.445	0.755	0.602
	Condor	0.637	0.875	0.875	0.230	0.374	0.800	0.486	0.567	0.435	0.751	0.603
	Magpie	0.629	0.869	0.875	0.125	0.352	0.798	0.482	0.553	0.436	0.742	0.586
	WizardLM	0.639	0.870	0.877	0.053	0.360	0.799	0.491	0.572	0.428	0.756	0.585
	Alpaca	0.634	0.871	0.880	0.042	0.362	0.803	0.495	0.580	0.445	0.753	0.586
	Self-Instruct	0.515	0.820	0.866	0.103	0.338	0.787	0.439	0.513	0.341	0.721	0.544
	Wiki	0.638	0.876	0.869	0.191	0.366	0.801	0.489	0.570	0.424	0.759	0.598
	Wiki-Rewrite	0.641	0.877	0.871	0.212	0.370	0.800	0.497	0.568	0.417	0.758	0.601
	Atomic-LIFD	0.671	0.886	0.884	0.364	0.420	0.800	0.523	0.568	0.398	0.751	0.626
	Atomic-QA	0.675	0.886	0.883	0.334	0.435	0.804	0.515	0.567	0.423	0.753	0.627

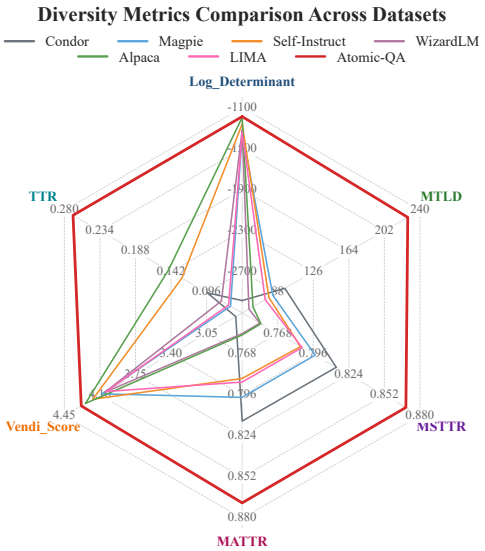


Figure 6: Radar plot of corpus diversity.

like WizardLM, Atomic-QA demonstrates consistent superiority. For instance, on the Qwen2.5-7B, Atomic-QA (61.1%) surpasses WizardLM (Xu et al., 2025) (56.8%) by over **4%**. Notably, Atomic-QA excels in reasoning-intensive tasks like OpenBookQA (Mihaylov et al., 2018) and ARC-Challenge (Clark et al., 2018), suggesting that preserving operator diversity is more critical.

6.2 Exclusion of Potentially Valuable Samples

Figure 6 visualizes corpus diversity using Vendi Score (Friedman and Dieng, 2023), Log Determinant (Yang et al., 2025b), TTR-variants (Covington and McFall, 2010), and MTLD (McCarthy and Jarvis, 2010). Atomic-QA achieves the highest scores across all six metrics and encompasses all baselines. While aggressively filtered datasets show contracted diversity. This confirms that tree-based operator enumeration preserves the structural samples that filtering discards. However, a practical question remains: what is the quantitative noise threshold in real-world synthetic data?

7 Generalization to Existing Datasets

To answer RQ3, we introduce the **Atomic-X** pipeline, which decomposes existing datasets into atomic operators and injects controlled perturbations. In right panel of Figure 2, Atomic-X adopts an Encoder–Perturber–Decoder pipeline:

Step 1 (Encoding): A encoder takes an instruction-response pair from an existing dataset and performs a top-down decomposition of the response into atomic operators, guided by the operator tree. The model recursively parses the content into distinct logical branches, where each leaf corresponds to an atomic operator in the tree, yielding an ordered sequence $\mathbb{A} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$.

Step 2 (Perturbing): A perturbation function is applied to the extracted atomic functions. As shown in the formula $\text{Perturbing}(\mathcal{A}_i) = \mathcal{A}_i'$, we inject operator-level noise (e.g., modifying operations from $1 + 1 = 2$ to $1 + 1 = 3$) to generate corrupted atomic functions.

Step 3 (Decoding): The decoder reconstructs

the output: First, the perturbed atomic functions are reranked based on the magnitude of their logical impact as $\mathbb{A}^* = \{\mathcal{A}_1^*, \mathcal{A}_2^*, \ldots, \mathcal{A}_n^*\}$. Next, the model generates specific decoding instructions for modification. Finally, a deterministic replace tool utilizes these instructions to edit the original text, producing the final *Perturbed Output*. **Further details are in Appendix B.2.** We apply Atomic-X to Alpaca (Taori et al., 2023) and MetaMath (Yu et al., 2023), producing **Atomic-Alpaca** and **Atomic-MetaMath**. This enables us to analyze noise tolerance on multi-step reasoning task.

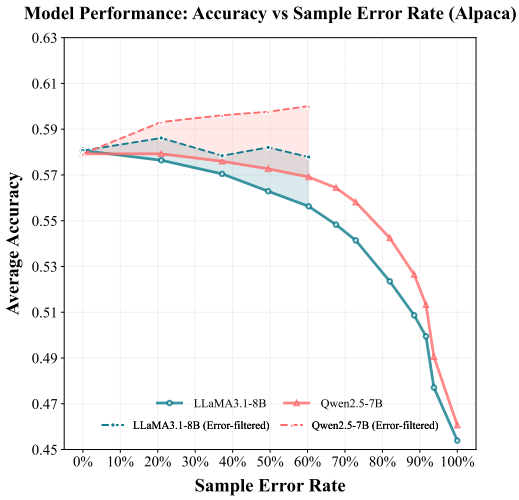


Figure 7: Average accuracy for LLaMA 3.1-8B and Qwen 2.5-7B on Atomic-Alpaca with operator perturbations at controlled rates (§7). Dashed lines: upper bound from training only on correct samples, truncated at 60% to approximate the solid lines' data scale. See Figure 17 for Atomic-MetaMath.

7.1 Operator Noise Tolerance Limits

To verify that operator sensitivity (§5) generalizes to real data, we adopt Beta-distribution-based sampling as described in Appendix E and inject operator erros through varying rates of atomic-operation perturbations into Alpaca and MetaMath. We analyze the impact of increasing atomic perturbations rates (from 0% to 100%) on model performance, visualized in Figure 7. Results reveal a distinct two-stage non-linear degradation pattern.

Stage 1: Gradual Logic Degradation: As shown in Figure 7, injecting perturbations into atomic operations results in a slow but observable performance decay during the initial stage ($0\% \rightarrow 40\%$ noise). Unlike operand, operator perturbations disrupt models' training process more severely, causing proportional capability loss.

Stage 2: Structural Collapse: As the error rate exceeds a critical threshold (roughly $> 60\%$ in Figure 7), the degradation pattern shifts distinctly, characterized by a significantly steepened negative slope. Visually, this is represented by the rapid plunge in accuracy towards the tail end of the curve ($60\% \rightarrow 100\%$). This sharp performance drop suggests that accumulated operator perturbations severely impair the model's ability to learn reliable patterns beyond this threshold.

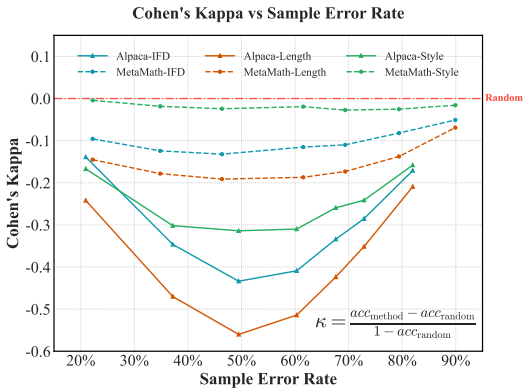


Figure 8: Cohen's Kappa between filtering methods (IFD, Length, Style) and ground-truth error labels as a function of sample error rate, evaluated on **Atomic-Alpaca** and **Atomic-MetaMath**.

7.2 Upper Bound vs. Filtering Reality

To assess filtering potential, we construct an oracle upper bound by discarding all samples containing operator errors based on ground-truth labels, and retraining on the remaining clean subset at each error rate (dashed lines in Figure 7). These oracle-filtered models fully recover the noise-free baseline, showing that operator-induced degradation is recoverable if erroneous samples can be identified. This sets the ceiling for any practical filter. Yet, as shown in Figure 8, existing strategies cannot approach this ceiling: their negative Cohen's Kappa scores indicate they are unable to differentiate between fault and correct samples.

8 Conclusion

This study quantifies error tolerance in synthesis data, moving beyond the binary high-low quality definitions. We demonstrate that SFT is fundamentally a logic-learning process in atomic view where models learn functional atomic operators while specific operand-values are changeable variables. We show that models learn atomic logic even from randomized operands. Thus, synthetic data should favor operators over operands for generalizability.

Limitation

Due to computational resource constraints, we do not conduct experiments on models with 32B parameters or larger. Furthermore, while our ATOM framework demonstrates robustness across general instruction following and mathematical reasoning tasks, we have not yet extended the atomic decomposition methodology to code generation or other domains, which we leave for future work.

References

Ahmed Alajrami, Xingwei Tan, and Nikolaos Aletras. 2025. Fine-tuning on noisy instructions: Effects on generalization and performance. *Preprint*, arXiv:2510.03528.

Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. *Preprint*, arXiv:1905.13319.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. Longbench: A bilingual, multitask benchmark for long context understanding. *Preprint*, arXiv:2308.14508.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2019. Piqa: Reasoning about physical commonsense in natural language. *Preprint*, arXiv:1911.11641.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *Preprint*, arXiv:2402.03216.

Runjin Chen, Gabriel Jacob Perin, Xuxi Chen, Xilun Chen, Yan Han, Nina S. T. Hirata, Junyuan Hong, and Bhavya Kailkhura. 2025a. Extracting and understanding the superficial knowledge in alignment. *Preprint*, arXiv:2502.04602.

Yicheng Chen, Yining Li, Kai Hu, Zerun Ma, Haochen Ye, and Kai Chen. 2025b. Mig: Automatic data selection for instruction tuning by maximizing information gain in semantic space. *Preprint*, arXiv:2504.13835.

Hyunsoo Cho. 2024a. Unveiling imitation learning: Exploring the impact of data falsity to large language model. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 62–73, Bangkok, Thailand. Association for Computational Linguistics.

Hyunsoo Cho. 2024b. Unveiling imitation learning: Exploring the impact of data falsity to large language model. *Preprint*, arXiv:2404.09717.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. Boolq: Exploring the surprising difficulty of natural yes/no questions. *Preprint*, arXiv:1905.10044.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *Preprint*, arXiv:1803.05457.

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. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

Michael A. Covington and Joe D. McFall. 2010. Cutting the gordian knot: The moving-average type–token ratio (mattr). *Journal of Quantitative Linguistics*, 17:100 – 94.

Qirun Dai, Dylan Zhang, Jiaqi W. Ma, and Hao Peng. 2025. Improving influence-based instruction tuning data selection for balanced learning of diverse capabilities. *Preprint*, arXiv:2501.12147.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Benoit Frenay and Michel Verleysen. 2014. Classification in the presence of label noise: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 25(5):845–869.

Dan Friedman and Adji Bousso Dieng. 2023. The vendi score: A diversity evaluation metric for machine learning. *Transactions on Machine Learning Research*.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, and 5 others. 2024. The language model evaluation harness.

Gemini-Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy Lillicrap, Angeliki Lazaridou, and 1332 others. 2025. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks are all you need. *Preprint*, arXiv:2306.11644.

Alex Havrilla and Maia Iyer. 2024. Understanding the effect of noise in llm training data with algorithmic chains of thought. *Preprint*, arXiv:2402.04004.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *Preprint*, arXiv:2103.03874.

Tingyu Jiang, Shen Li, Yiyao Song, Lan Zhang, Hualei Zhu, Yuan Zhao, Xiaohang Xu, Kenjiro Taura, and Hao Henry Wang. 2025. Importance-aware data selection for efficient llm instruction tuning. *Preprint*, arXiv:2511.07074.

Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. 2019. Similarity of neural network representations revisited. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3519–3529. PMLR.

Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. 2017. Race: Large-scale reading comprehension dataset from examinations. *Preprint*, arXiv:1704.04683.

Ming Li, Yong Zhang, Shwai He, Zhitao Li, Hongyu Zhao, Jianzong Wang, Ning Cheng, and Tianyi Zhou. 2024. Superfiltering: Weak-to-strong data filtering for fast instruction-tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14255–14273, Bangkok, Thailand. Association for Computational Linguistics.

Zhuang Li, Yuncheng Hua, Thuy-Trang Vu, Haolan Zhan, Lizhen Qu, and Gholamreza Haffari. 2025. SCAR: Data selection via style consistency-aware response ranking for efficient instruction-tuning of large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12756–12790, Vienna, Austria. Association for Computational Linguistics.

Bill Yuchen Lin, Abhilasha Ravichander, Ximing Lu, Nouha Dziri, Melanie Sclar, Khyathi Chandu, Chandra Bhagavatula, and Yejin Choi. 2023. The unlocking spell on base llms: Rethinking alignment via in-context learning. *Preprint*, arXiv:2312.01552.

Cao Maosong, Taolin Zhang, Mo Li, Chuyu Zhang, Yunxin Liu, Conghui He, Haodong Duan, Songyang Zhang, and Kai Chen. 2025. Condor: Enhance LLM alignment with knowledge-driven data synthesis and refinement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 22392–22412, Vienna, Austria. Association for Computational Linguistics.

Philip M McCarthy and Scott Jarvis. 2010. Mtld, vocd-d, and hd-d: A validation study of several new measures of lexical diversity. *Behavior Research Methods*, 42(2):381–392.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? a new dataset for open book question answering. *Preprint*, arXiv:1809.02789.

Allen Newell. 1972. *Human Problem Solving*. Prentice-Hall, Inc., USA.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Qwen-Team. 2024. Qwen2.5: A party of foundation models.

Mohit Raghavendra, Vaskar Nath, and Sean M. Hendryx. 2025. Revisiting the superficial alignment hypothesis.

Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for squad. *Preprint*, arXiv:1806.03822.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. Winogrande: An adversarial winograd schema challenge at scale. *Preprint*, arXiv:1907.10641.

Maarten Sap, Hannah Rashkin, Derek Chen, Ronan LeBras, and Yejin Choi. 2019. Socialiqa: Commonsense reasoning about social interactions. *Preprint*, arXiv:1904.09728.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607. Springer.

Ozan Sener and Silvio Savarese. 2018. Active learning for convolutional neural networks: A core-set approach. In *International Conference on Learning Representations*.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

Van-Hien Tran, Hiroki Ouchi, Taro Watanabe, and Yuji Matsumoto. 2022. Improving discriminative learning for zero-shot relation extraction. In *Proceedings of the 1st Workshop on Semiparametric Methods in NLP: Decoupling Logic from Knowledge*, pages 1–6, Dublin, Ireland and Online. Association for Computational Linguistics.

Peng Wang, Ningyu Zhang, Xin Xie, Yunzhi Yao, Bozhong Tian, Mengru Wang, Zekun Xi, Siyuan Cheng, Kangwei Liu, Guozhou Zheng, and 1 others. 2023a. Easyedit: An easy-to-use knowledge editing framework for large language models. *arXiv preprint arXiv:2308.07269*.

Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, Jinsong Su, Qi Chen, and Scarlett Li. 2025. Epicoder: Encompassing diversity and complexity in code generation. In *Forty-second International Conference on Machine Learning*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023b. Self-instruct: Aligning language models with self-generated instructions. *Preprint*, arXiv:2212.10560.

Yong Xie, Karan Aggarwal, Aitzaz Ahmad, and Stephen Lau. 2024. Controlled automatic task-specific synthetic data generation for hallucination detection. *Preprint*, arXiv:2410.12278.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2025. Wizardlm: Empowering large pre-trained language models to follow complex instructions. *Preprint*, arXiv:2304.12244.

Zhangchen Xu, Fengqing Jiang, Luyao Niu, Yuntian Deng, Radha Poovendran, Yejin Choi, and Bill Yuchen Lin. 2024. Magpie: Alignment data synthesis from scratch by prompting aligned llms with nothing. *Preprint*, arXiv:2406.08464.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Yuming Yang, Yang Nan, Junjie Ye, Shihan Dou, Xiao Wang, Shuo Li, Huijie Lv, Tao Gui, Qi Zhang, and Xuanjing Huang. 2025b. Measuring data diversity for instruction tuning: A systematic analysis and a reliable metric. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 18530–18549, Vienna, Austria. Association for Computational Linguistics.

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

Hao Zhao, Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. 2024. Long is more for alignment: A simple but tough-to-beat baseline for instruction fine-tuning. In *Forty-first International Conference on Machine Learning*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024a. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, Bangkok, Thailand. Association for Computational Linguistics.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024b. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023. Lima: Less is more for alignment. *Preprint*, arXiv:2305.11206.

A Expanded Related Work

Construction and Filtering of High-Quality Data for SFT Early work highlights the importance of training data quality over sheer data scale in SFT. LIMA (Zhou et al., 2023) and Textbooks Are All You Need (Gunasekar et al., 2023) demonstrate that effective alignment does not strictly require large-scale datasets, but can emerge from relatively small yet high-quality training corpora. In a similar spirit, generate-then-filter pipelines such as Self-Instruct (Wang et al., 2023b), WizardLM (Xu et al., 2025), and Magpie (Xu et al., 2024) explore scalable, high-quality data construction through one or more rounds of data generation and heuristic filtering. More recently, Condor (Maosong et al.,

2025) refines synthetic training data during generation by constructing instructions of varying difficulty and improving response quality.

Other work in this line focuses on filtering and selecting representative high-quality training data. Many approaches rely on surface-level quality metrics or heuristic-based strategies, including length-based filtering (Zhao et al., 2024), perplexity-based selection (Li et al., 2024), and style-consistency constraints (Li et al., 2025). More recent work has proposed methods that estimate sample contributions to downstream performance, such as BIDS (Dai et al., 2025), MIG (Chen et al., 2025b), and MIWV (Jiang et al., 2025).

However, most data synthesis, refinement, and selection methods lack a guiding framework for quantitatively defining data quality. Consequently, they oscillate between two extremes: overly aggressive filtering that removes valuable supervisory signals, and overly permissive filtering that retains training instances with potential errors.

Training Data Noise Effects and Learning Mechanisms in SFT Prior work has investigated noisy supervision in SFT under a variety of controlled settings. FTNI (Alajrami et al., 2025) observes that training on perturbed instructions (such as removing stop words or shuffling words) can improve downstream performance in some cases. For reasoning supervision, TInt (Havrilla and Iyer, 2024) studies robustness to noise in algorithmic chains of thought by injecting errors during reasoning, and FACO (Cho, 2024b) studies how different proportions of erroneous reasoning-chain samples in the training data affect model performance.

From a learning-mechanism perspective, the superficial alignment hypothesis characterizes SFT as primarily focusing on adopting the language style of responsible AI assistants and relying largely on the knowledge already acquired by base LLMs (Zhou et al., 2023; Lin et al., 2023). Beyond this view, subsequent evidence shows that SFT enhances models' reasoning and contextual understanding compared to their base counterparts (Chen et al., 2025a). Raghavendra et al. (2025) further observe that, similar to the pre-training scaling laws, post-training task performance scales as a power law against the number of finetuning examples, reflecting that language models are not necessarily confined to using only the knowledge learned during pretraining.

Current analyses of noise in training data and SFT learning mechanisms remain confined to specific types of datasets or lack fine-grained, sample-level structural analysis. As a result, they are unable to determine which errors are tolerable to the model and which genuinely disrupt learning, leaving the critical boundary between benign and detrimental noise unclear and the quantification of error tolerance in synthetic data unexplored.

B Pipeline for ATOM

This section presents the detailed process of ATOM. We elaborate from the perspectives of the Pipeline for Atomic Tree Construction and Atomic Data Synthesis, respectively.

B.1 Pipeline for Atomic Tree Construction

To facilitate clarity, we first detail the prompts employed in our pipeline, followed by a formal description using pseudo-code.

ATOM initiates the process by extracting atomic operators from representative data, guided by predefined definitions via Prompt 1. Subsequently, we organize these operators into an Initial Operator Tree using Prompt 2, where internal nodes represent abstract operator categories and leaf nodes represent concrete atomic operators. To guarantee structural completeness, we generate trees for various subsets of atomic functions under diverse sampling schemes, while simultaneously pruning duplicate nodes at the same hierarchy level via Prompt 3. After the above steps, we obtain the operator trees shown in Figure 9, 10 and 11. Finally, building upon this initial structure and its enumerated first-layer nodes, we expand the tree across both depth and breadth dimensions using Prompt 4. The comprehensive procedure for Atomic Tree Construction is outlined in Algorithm 1.

Human Expert Verification. To ensure the quality and validity of the constructed atomic trees, we involve four human experts throughout the pipeline. During **Initial Tree Generation** (Phase 2), the experts evaluate whether the category hierarchy is *reasonable* (i.e., semantically coherent groupings) and *comprehensive* (i.e., sufficient coverage of the operator space). An initial tree is accepted only when all four experts reach unanimous agreement; otherwise, the tree is revised and re-evaluated. During **Tree Expansion** (Phase 3), the experts verify whether the newly generated leaf nodes conform to the atomic operator definition in our formulation—namely, that each leaf represents a semantically

Algorithm 1: Atomic Tree Construction Pipeline

Input : Candidate Datasets $\mathcal{D}$, Embedding Model $\mathcal{M}_{emb}$, Generation Model $\mathcal{M}_{gen}$, Max Interaction $\mathcal{I}$
Output : Atomic Tree $\mathcal{T}$

```
// Phase 1: Representative Data Selection
V ← Encode(D, M_bge),
S_rep ← KCenterGreedy(V),
T ← {root}
A_rep ← Extract(S_rep)
// Phase 2: Level 1 - Initial Tree Generation
for batch A_sub ∈ sample(A_rep) do
    T_sub ← GenSubTree(A_sub, M_gen)
    T ← Merge(T, T_sub)
// Phase 3: Level 2 - Tree Expansion
for iter i ∈ I do
    T_i ← ∅
    for node n ∈ LeafNodes(T) do
        N_d ← DepthExpansion(n, M_gen)
        N_b ← BreadthExpansion(n, M_gen)
        if N_d ≠ ∅ ∨ N_b ≠ ∅ then
            T_i ← Merge(T_i, N_d ∪ N_b)
    if T_i = ∅ then
        break
    T ← Merge(T, T_i)
return T
```

irreducible operator. We compare the expert judgments with the LLM-based assessment and observe an agreement rate of 97.3%, indicating strong alignment between model and human evaluation.

B.2 Pipeline for Atomic Data Synthesis

This section details the Atomic-QA Generation pipeline. The Atomic-X Generation pipeline is described in §7 of the main text.

B.2.1 Atomic-QA Generation

Atomic-QA generation follows the pipeline described in §3.2. Starting from each leaf node of the operator tree, we first employ an LLM to generate diverse operands x within the operator's domain, guided by Prompt 5. We then compute the corresponding outputs $y = f(x)$ via a subsequent LLM call to form complete atomic triplets $\mathcal{A} = \langle x, f, y \rangle$. Finally, we utilize Prompt 6 to validate the logical correctness of the generated atomic operations.

B.2.2 Atomic-X Generation

The Atomic-X pipeline is introduced in §7; here we detail the prompt usage. The framework operates in the reverse direction: given existing data, it decomposes responses into atomic operators. The framework comprises three core modules: the Encoder, the Perturber, and the Decoder. Guided by Prompt 7, the Encoder maps the response content onto the operator tree, recursively decomposing the structure until reaching leaf operators, which are resolved into atomic operations. Next, the Perturber applies operator perturbations to each atomic operation utilizing the Prompt 8. Finally, the Decoder modifies the original data by employing the SEARCH-REPLACE and TERMINATE tools defined in Prompt 9.

C Details about Experimental Setup

In this section we will introduce our experimental details. First, we introduce the details of data generation used in the experiments, followed by a description of the model training and evaluation procedures. **All experiments are run three times.**

C.1 Details about Dataset Generation

For the Atomic-QA generation process, we commence by employing the KCenterGreedy algorithm to identify 5,000 cluster centers. The initial operator tree is constructed through an iterative procedure spanning 500 rounds, with 50 subjects randomly sampled in each round. During the subsequent evolution stage, the tree evolves for 1,715 iterations. Notably, in each iteration, a single leaf operator node undergoes five depth expansions and five breadth expansions. This results in an operator tree with 6,931 leaf nodes (max depth 4), where each leaf defines a concrete atomic operator. Following the pipeline in §3.2, we generate diverse operands for each leaf operator, yielding the Atomic-QA dataset with approximately 1.37 million instances.

Regarding the Atomic-X generation process, we construct a separate operator tree for each target dataset. We initiate construction by randomly sampling 10,000 instances, generating 100 cluster centers at each step. The initial tree construction involves 200 evolution rounds, wherein 50 data points are selected per round. In the final

Figure 9: The initial operator tree of Atomic-QA. Internal nodes represent abstract operator categories, and leaf nodes define concrete atomic operators.

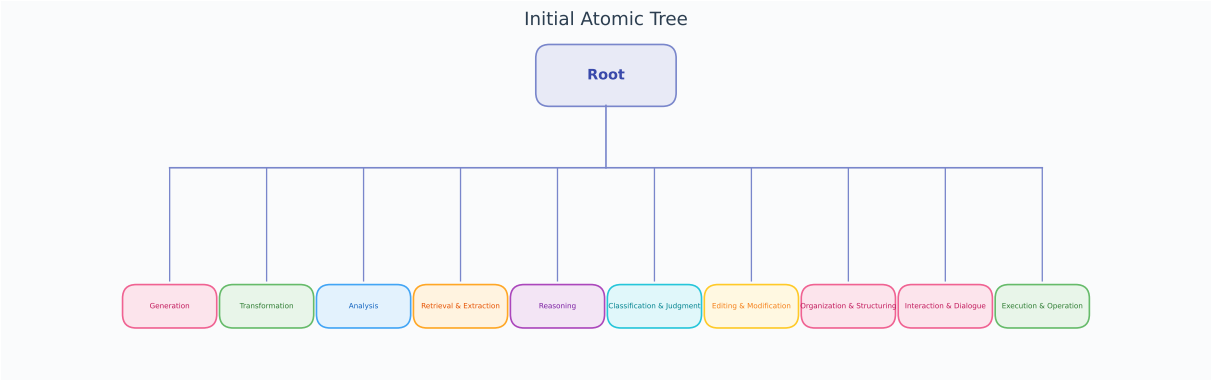


Figure 10: The initial operator tree of Alpaca.

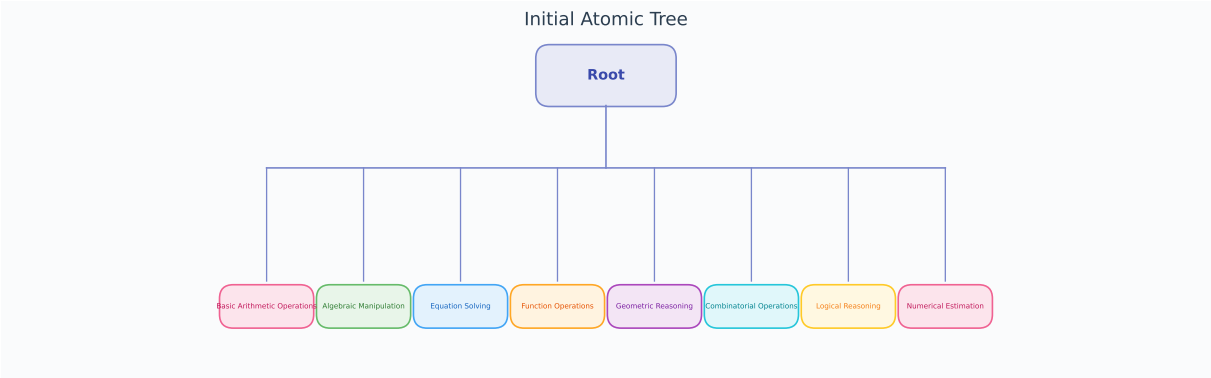


Figure 11: The initial operator tree of MetaMath.

stage, the evolution iterations differ by dataset: Alpaca evolves for 1,203 iterations, while MetaMath evolves for 1,347 iterations. This results in operator trees with 276 and 398 leaf operators respectively for Alpaca (max depth 5) and MetaMath (max depth 6). From the remaining samples, we randomly select 10,000 instances and apply the Atomic-X pipeline (§7) to construct the corresponding Atomic-Alpaca and Atomic-MetaMath.

C.2 Details about Model Training and Evaluation

We conduct experiments on four representative open-source pre-trained LLMs that span differ-

ent architectures and scales, including LLaMA3.1-8B, Qwen3-8B, Qwen2.5-7B, and Qwen2.5-14B. All models are fine-tuned with LLaMA-Factory (Zheng et al., 2024a) and evaluated with the LM Evaluation Harness (Gao et al., 2024).

Datasets. We compare models fine-tuned separately under different training dataset configurations, as summarized in Tables 1 and 3. The comparison set includes Atomic-QA and its variants (Wiki, Wiki-Rewrite, and Atomic-LIFD), the official *Instruct* versions, and six representative open-source datasets: LIMA (Zhou et al., 2023), Condor (Maosong et al., 2025), Magpie (Xu et al., 2024), WizardLM (Xu et al., 2025), Alpaca (Taori et al., 2023), and Self-Instruct (Wang et al., 2023b). Specifically, *Instruct* refers to the official tuned versions released for each pretrained model. *Wiki* consists of synthetic instructions paired with Wikipedia entity summaries as responses, where the entities are drawn from the Atomic-QA entity set with available Wikipedia pages. *Wiki-Rewrite* reformulates these instructions to increase diversity, serving as a variant derived from Wiki. In addition, *Atomic-LIFD* is a compact distilled subset of Atomic-QA obtained via perplexity-based selection, with construction details provided in Appendix D.1.

Training Configuration. All models are fine-tuned using Low-Rank Adaptation (LoRA) with rank $r = 16$ and a weight decay of 0.01 for 2 epochs. We adopt a learning rate of 1×10^{-4} by default. For experiments involving scaled training data, specifically Atomic-QA MCQ, perplexity analysis, and CKA variance analysis, we use a learning rate of 1×10^{-5}, as shown respectively in Figure 4, Figure 12, and Figure 13.

Evaluation Configuration. We evaluate Atomic-QA alongside other dataset baselines across a benchmark suite spanning multiple capability dimensions. In the domain of text comprehension and information integration, assessment is conducted using BoolQ (Clark et al., 2019), LongBench-HotpotQA (Bai et al., 2024), RACE (Lai et al., 2017), and SQuADv2 (Rajpurkar et al., 2018). With regard to knowledge-driven reasoning, capabilities are measured through ARC-Easy/Challenge (Clark et al., 2018) and OpenBookQA (Mihaylov et al., 2018). Additionally, for commonsense reasoning, evaluation is performed via PIQA (Bisk et al., 2019), SIQA (Sap et al., 2019), and WinoGrande (Sakaguchi et al., 2019).

Table 2: Evaluation benchmarks and configurations.

Category	Benchmark	Shots	Metric
Text Comprehension and Information Integration	BoolQ	3	Accuracy
	LongBench-HotpotQA	0	F1
	RACE	3	Accuracy
	SQuADv2	3	F1
Knowledge-Driven Reasoning	ARC-Challenge	3	Accuracy
	ARC-Easy	3	Accuracy
	OpenBookQA	3	Accuracy
Commonsense Reasoning	PIQA	3	Accuracy
	SIQA	3	Accuracy
	WinoGrande	3	Accuracy
Mathematical Reasoning	GSM8K	3	Exact Match
	Hendrycks Math500	3	Exact Match
	MathQA	3	Accuracy

To analyze the impact of increasing atomic error rates (from 0% to 100%) on model performance, we evaluate datasets generated via the Atomic-X pipeline. Specifically, Atomic-Alpaca is tested on the identical ten benchmarks introduced above to observe how different error levels influence general capability dimensions. In parallel, Atomic-MetaMath is evaluated on specialized mathematical reasoning tasks, including GSM8K (Cobbe et al., 2021), Math500 (Hendrycks et al., 2021), and MathQA (Amini et al., 2019). Regarding the evaluation protocol, the majority of benchmarks are conducted under a 3-shot setting, while LongBench-HotpotQA and Atomic-QA MCQ follow a zero-shot paradigm. Results on Atomic-QA MCQ are reported in Figure 4, and detailed evaluation configurations for all benchmarks are summarized in Table 2.

D Additional Results

In this section, we present additional experimental results, structured to mirror the organization of the main text. We begin with supplementary findings on Quantifying the Cost of Strict Filtering, followed by further details on Measuring Operator–Operand and Dominance.

D.1 Training Tokens Comparison

Table 4 reports the total number of training tokens for each dataset. The datasets vary substantially in scale, ranging from fewer than one million tokens (e.g., LIMA) to over two hundred million tokens (e.g., Magpie), with several datasets clustered in the range of a few million to several tens of millions of tokens. Given this substantial variation, it is important to determine whether dataset scale or structural diversity serves as the primary determinant of performance. To investigate this question, we con-

Table 3: Overall performance across all training datasets on Qwen2.5-7B and Qwen2.5-14B. Color-coded cells (green/red) are used to indicate performance above or below the mean across methods (All Average), with darker colors representing better or worse performance.

Model Family		ARC-Challenge	ARC-Easy	BoolQ	Hotpot-QA	Open-BookQA	PIQA	RACE	SIQA	SQuADv2	Wino-Grande	Avg
Qwen2.5-7B	Base	0.590	0.860	0.872	0.216	0.390	0.803	0.459	0.576	0.421	0.754	0.594
	All Average	0.586	0.859	0.875	0.190	0.399	0.800	0.474	0.572	0.364	0.747	0.587
	Instruct	0.616	0.873	0.865	0.143	0.404	0.786	0.512	0.566	0.324	0.735	0.582
	LIMA	0.586	0.861	0.871	0.207	0.396	0.801	0.459	0.574	0.424	0.757	0.594
	Condor	0.597	0.864	0.874	0.185	0.392	0.801	0.471	0.580	0.396	0.755	0.591
	Magpie	0.582	0.857	0.876	0.241	0.364	0.795	0.455	0.572	0.412	0.740	0.589
	WizardLM	0.606	0.863	0.879	0.050	0.412	0.811	0.464	0.584	0.259	0.756	0.568
	Alpaca	0.586	0.862	0.883	0.037	0.392	0.808	0.482	0.587	0.265	0.747	0.565
	Self-Instruct	0.490	0.816	0.859	0.062	0.376	0.792	0.430	0.524	0.304	0.723	0.537
	Wiki	0.583	0.868	0.876	0.248	0.398	0.803	0.470	0.576	0.426	0.763	0.601
	Wiki-Rewrite	0.584	0.870	0.876	0.204	0.404	0.802	0.470	0.576	0.422	0.751	0.596
	Atomic-LIFD	0.606	0.859	0.882	0.383	0.432	0.797	0.500	0.576	0.392	0.743	0.617
	Atomic-QA	0.609	0.860	0.880	0.327	0.419	0.801	0.505	0.577	0.386	0.746	0.611
Qwen2.5-14B	Base	0.637	0.878	0.888	0.147	0.404	0.822	0.487	0.579	0.392	0.793	0.603
	All Average	0.630	0.879	0.884	0.169	0.409	0.816	0.508	0.586	0.393	0.791	0.607
	Instruct	0.699	0.902	0.883	0.160	0.430	0.821	0.577	0.611	0.411	0.785	0.628
	LIMA	0.634	0.880	0.886	0.139	0.404	0.820	0.484	0.582	0.395	0.800	0.602
	Condor	0.631	0.879	0.885	0.133	0.396	0.818	0.490	0.588	0.379	0.800	0.600
	Magpie	0.630	0.882	0.893	0.037	0.368	0.811	0.496	0.580	0.415	0.776	0.589
	WizardLM	0.640	0.888	0.893	0.076	0.400	0.824	0.520	0.590	0.351	0.789	0.597
	Alpaca	0.609	0.880	0.898	0.143	0.394	0.822	0.500	0.597	0.427	0.796	0.607
	Self-Instruct	0.518	0.826	0.850	0.263	0.368	0.800	0.463	0.534	0.319	0.763	0.570
	Wiki	0.629	0.880	0.874	0.068	0.416	0.817	0.508	0.597	0.417	0.807	0.601
	Wiki-Rewrite	0.626	0.880	0.882	0.063	0.408	0.820	0.515	0.592	0.416	0.811	0.601
	Atomic-LIFD	0.653	0.878	0.894	0.414	0.466	0.808	0.507	0.582	0.392	0.770	0.636
	Atomic-QA	0.666	0.889	0.880	0.367	0.449	0.817	0.539	0.592	0.405	0.803	0.641

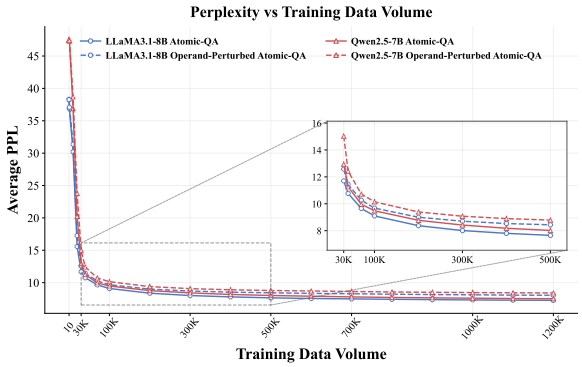


Figure 12: Perplexity trends of LLaMA and Qwen models as a function of training data volume. The curve displays average PPL for both the original and entity-replaced variants, with an inset showing a zoomed region for the 30K–500K data range.

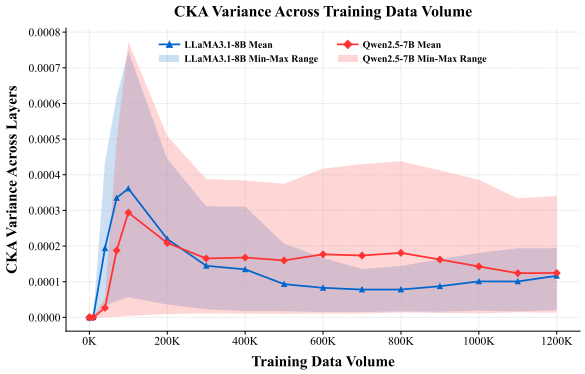


Figure 13: CKA variance across layers versus training data volume for LLaMA3.1-8B and Qwen2.5-7B, showing mean values (lines) and min–max ranges (shaded areas).

Table 4: The number of training tokens in different datasets.

Dataset	Training Tokens
LIMA	627,993
Condor	21,131,550
Magpie	203,256,865
WizardLM	27,673,025
Alpaca	3,766,455
Self-Instruct	4,536,225
Wiki	11,082,377
Wiki-Rewrite	11,356,362
Atomic-QA	34,901,525
Atomic-LIFD	**2,701,510**

struct Atomic-LIFD, a compact dataset of approximately 2.7M tokens that is larger than LIMA but smaller than other considered datasets. Specifically, this subset is derived by applying perplexity-based selection (IFD) to Atomic-QA, a logically verified and structurally diverse source.

Remarkably, Atomic-LIFD achieves performance comparable to that of the full Atomic-QA dataset, as shown in Table 1 and Table 3. Given that data scale is the sole variable distinguishing these two settings, this minimal performance gap strongly suggests that data quantity is not the decisive factor behind Atomic-QA's superiority.

D.2 Quantifying the Cost of Strict Filtering

Table 3 extends our evaluation to Qwen2.5-7B and Qwen2.5-14B to verify the robustness of our approach. Consistent with the findings in the main

text, Atomic-QA demonstrates strong generalizability, achieving the highest average performance across all baselines on both backbones (61.2% on Qwen2.5-7B and 62.5% on Qwen3-8B).

The persistent performance gap over LIMA (+1.8% to +2.3%) and other synthetic methods (Alpaca, WizardLM) confirms that our conclusion is model-agnostic: prioritizing structural scale over strict filtering consistently yields better instruction-following and reasoning capabilities, regardless of the underlying model architecture or size.

D.3 Measuring Operator–Operand and Dominance

We further validated our findings on the Qwen model family, as illustrated in Figures 14, 15, and 16. Across varying model sizes (7B, 8B, and 14B), Operand-Perturbed Atomic-QA consistently aligns with standard Atomic-QA performance with negligible degradation (e.g., an average drop of only 0.002 for Qwen3-8B). This stability demonstrates that robustness to operand errors is a generalizable phenomenon, independent of specific model architectures.

To substantiate our main findings, we visualize the training dynamics in Figures 12 and 13. Figure 12 shows that the perplexity curves for the Operand-Perturbed Atomic-QA and the original dataset are nearly indistinguishable, demonstrating that operand errors impose almost no additional optimization burden. This indicates the model learns the structural logic with equal efficiency. Furthermore, Figure 13 illustrates that the variance in CKA similarity initially spikes but subsequently stabilizes and decreases as data volume increases. This convergence confirms that the high representational alignment discussed in the main text is a robust and stable outcome of training, rather than a transient state.

Finally, to verify the generalizability of the degradation patterns in operation errors, we extended our analysis to mathematical reasoning using the MetaMath dataset. As illustrated in Figure 17, consistent with the collapse observed in Alpaca, both LLaMA 3.1-8B and Qwen 2.5-7B eventually exhibit a sharp performance decline. However, a key distinction lies in the critical threshold. While models trained on Alpaca begin to collapse around an error rate of 0.6, those trained on MetaMath demonstrate higher robustness, maintaining stability until the error rate approaches saturation (> 0.9). This confirms that while the non-linear degradation dynamics are universal across tasks, the tipping point for structural collapse depends on domain complexity. Overall, the models exhibit exceptional resilience to operand perturbations. Remarkably, even when the operand error rate reaches almost 100%, performance remains largely unaffected. In stark contrast, a 100% error rate in operators precipitates a significant performance decline across both the Alpaca and MetaMath datasets.

Table 5: Average perplexity of training corpora evaluated by pretrained checkpoints.

Model	Atomic-QA	Operator-Pert.	Operand-Pert.
LLaMA3.1-8B	35.80	36.53	99.76
Qwen3-8B	36.00	33.04	90.78
Qwen2.5-7B	43.33	42.17	136.24
Qwen2.5-14B	52.25	39.48	131.47

D.4 Surface-Level Coherence of Operator vs. Operand Perturbations

A natural question is how operator/operand perturbation relates to the distinction between coherent but non-factual and incoherent text. We examine this by measuring textual coherence via perplexity, computed by base models, as shown in Table 5.

Operator-Perturbed data shows perplexity nearly identical to Atomic-QA like 35.80 vs. 36.53 on LLaMA3.1-8B, indicating high surface fluency despite logical corruption. Operand-Perturbed data, by contrast, yields roughly 3× higher perplexity (99.76), reflecting strong surface incoherence caused by random-string operands. This exposes a counterintuitive asymmetry: the surface-incoherent operand variant is benign for training, whereas the surface-coherent operator variant is catastrophic.

D.5 Statistical Significance

We perform paired Wilcoxon signed-rank tests across the 10 evaluation benchmarks, comparing the proposed Atomic-QA against the mean performance of all baseline methods on each model family. The improvements are statistically significant in every case ($p < 0.01$). On LLaMA 3.1-8B, Atomic-QA raises the average score from 0.5716 to 0.5980 (Δ=+0.0264, W=0.0, p=0.00195). On Qwen2.5-7B, the gain is +0.0313 (0.5805→0.6118, W=0.0, p=0.00195). The larger Qwen2.5-14B also benefits substantially (Δ=+0.0377, W=3.0, p=0.00977), as does Qwen3-8B (Δ=+0.0375, W=1.0, p=0.00391). These results confirm that the advantages of Atomic-QA over existing data synthesis and selection strategies are robust and not

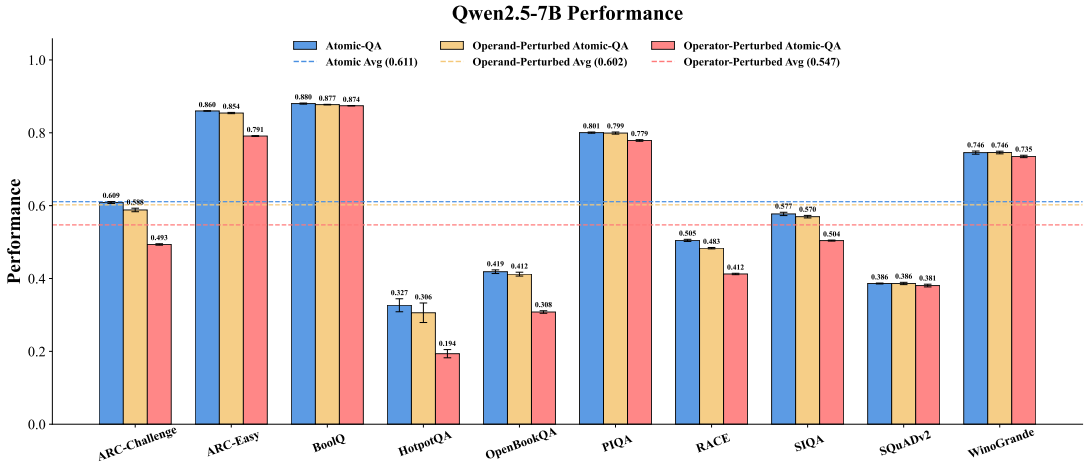


Figure 14: Performance comparison of Qwen2.5-7B across the Atomic-QA, Operand-Perturbed Atomic-QA, and Operator-Perturbed Atomic-QA settings. Each bar shows the model's accuracy on individual tasks, while the dashed lines denote the average performance for each setting.

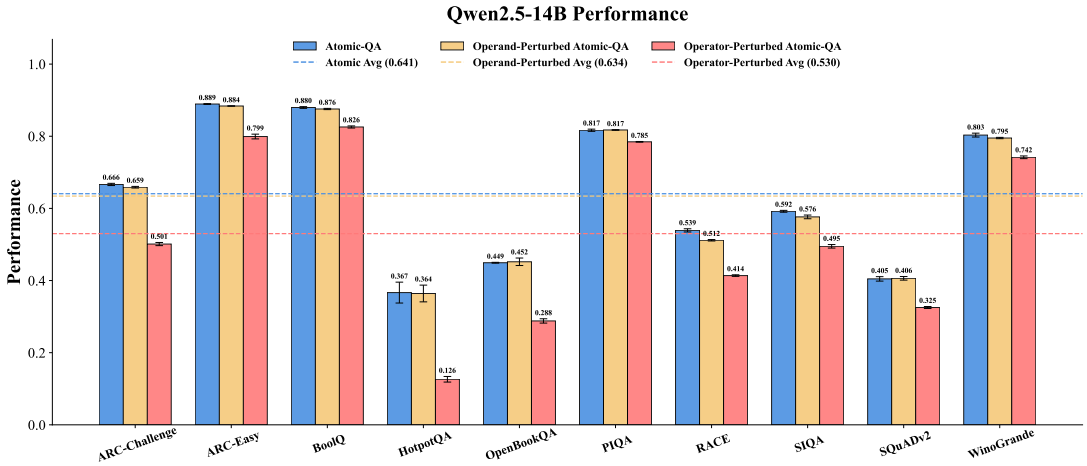


Figure 15: Performance comparison of Qwen2.5-14B across the Atomic-QA, Operand-Perturbed Atomic-QA, and Operator-Perturbed Atomic-QA settings. Each bar shows the model's accuracy on individual tasks, while the dashed lines denote the average performance for each setting.

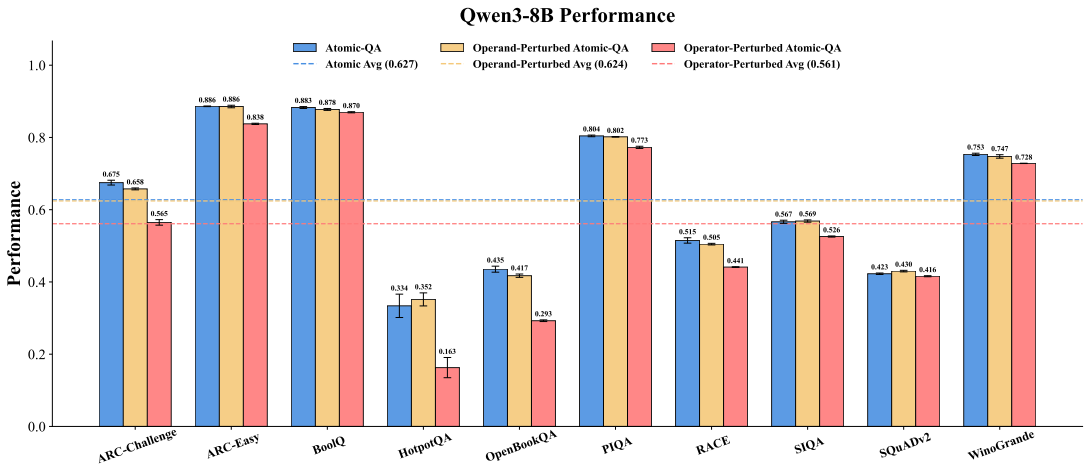


Figure 16: Performance comparison of Qwen3-8B across the Atomic-QA, Operand-Perturbed Atomic-QA, and Operator-Perturbed Atomic-QA settings. Each bar shows the model's accuracy on individual tasks, while the dashed lines denote the average performance for each setting.

Table 6: Performance comparison between Atomic-LIFD-20K and LIMA at matched data scale.

Training Data	Tokens	Model	Avg. Perf.
Atomic-LIFD-20K	535,706	LLaMA3.1-8B	0.583 (↑0.018)
		Qwen3-8B	0.627 (↑0.025)
		Qwen2.5-7B	0.623 (↑0.029)
		Qwen2.5-14B	0.638 (↑0.036)
LIMA	627,993	LLaMA3.1-8B	0.565
		Qwen3-8B	0.602
		Qwen2.5-7B	0.594
		Qwen2.5-14B	0.602

Table 7: Diversity metrics of Atomic-QA and its LIFD-filtered subsets.

Dataset	TTR	MTLD	MSTTR	MATTR	Vendi	LogDet
Atomic-QA	0.2688	226.97	0.8689	0.8695	4.2840	−1187.36
Atomic-LIFD-20K	0.2671	263.11	0.8817	0.8817	4.2963	−1169.71
Atomic-LIFD-100K	0.2616	253.22	0.8820	0.8806	4.2854	−1182.44

attributable to chance variation.

D.6 Comparison at Matched Data Scale

To investigate whether the advantages of our method persist under a controlled data-scale comparison, we construct **Atomic-LIFD-20K**, a more aggressively filtered subset of Atomic-QA designed to match the token budget of LIMA (∼600K tokens). In Table 6, Atomic-LIFD-20K consistently outperforms LIMA across all model architectures, despite using approximately 15% fewer tokens.

D.7 Diversity Retention under LIFD Filtering

To assess how much corpus diversity is retained after LIFD-based filtering, we compute diversity metrics for Atomic-QA, Atomic-LIFD-20K, and Atomic-LIFD-100K. As shown in Table 7, the filtered subsets preserve comparable or even higher diversity than the full dataset, demonstrating that LIFD effectively retains diverse samples while reducing data volume.

D.8 Representation Stability of LIMA

To investigate whether the high CKA similarity between LIMA-finetuned models and their base counterparts (Figure 13) is an artifact of under-training (e.g., insufficient epochs or an unadjusted learning rate schedule), we extend the training of LIMA from 2 to 5 epochs and measure CKA similarity relative to the base model at each stage. As shown in Table 8, the marginal decrease in CKA across extended epochs indicates that the models have effectively converged and are not significantly drifting from the base model's representations.

Table 8: CKA similarity between LIMA-finetuned and base models across training epochs.

Model	Epoch 2	Epoch 3	Epoch 4	Epoch 5
LLaMA-3.1-8B	0.9995	0.9984	0.9961	0.9922
Qwen2.5-7B	0.9996	0.9986	0.9970	0.9944

Table 9: Average performance of LIMA-finetuned models across training epochs.

Model	Epoch 2	Epoch 3	Epoch 4	Epoch 5
LLaMA-3.1-8B	0.565	0.563	0.566	0.564
Qwen2.5-7B	0.594	0.592	0.601	0.594

We further evaluate the downstream performance to check for potential gains from extended training. As shown in Table 9, the average performance saturates after epoch 2, with negligible improvements in subsequent epochs, confirming that the training has converged and additional epochs do not yield meaningful representation changes.

E Beta-Distribution Based Error Assignment

In this section, we present a Beta-Distribution based approach for assigning errors to samples while maintaining a target global error rate.We first formulate the problem, then introduce our stochastic assignment method that leverages the Beta distribution to create realistic error diversity across samples.

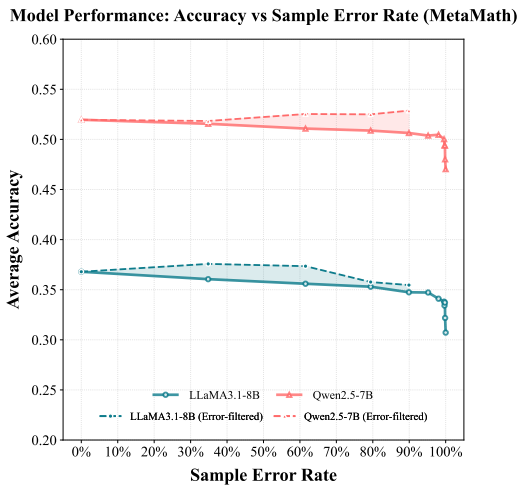


Figure 17: Accuracy on GSM8K, MATH-500, and MathQA for LLaMA 3.1-8B and Qwen 2.5-7B fine-tuned on **Atomic-MetaMath** with operator-level perturbations at controlled rates. Corresponds to Figure 7, which reports Atomic-Alpaca on general benchmarks.

E.1 Problem Formulation

We consider a dataset of M samples, where sample i comprises n_i atomic functions. Our objective is to introduce errors while matching a prescribed global error rate $r \in [0, 1]$ at the atomic-function level. The total number of atomic functions is $N = \sum_{i=1}^{M} n_i$.

For each sample i, we pre-generate $n_i + 1$ candidate versions indexed by $k \in \{0, 1, \ldots, n_i\}$, where the k-th version contains exactly k corrupted functions. Each version is constructed by introducing one additional corruption relative to the $(k-1)$-th version. Exactly one version must be selected for each sample to satisfy the following condition:

$$\sum_{i=1}^{M} e_i = \text{round}(r \cdot N),$$

where e_i is the number of errors in the selected version of sample i, and round$(\cdot)$ rounds to the nearest integer.

E.2 Beta Distribution Based Assignment

Motivation. A naive uniform approach assigns errors deterministically by setting $e_i = \lfloor r \cdot n_i \rfloor$ for each sample. While this guarantees the global error rate, it produces homogeneous error distributions where every sample exhibits approximately the same error rate. Such uniformity fails to capture the natural heterogeneity observed in real-world data, where some samples are nearly perfect while others contain multiple errors. To address this limitation, we introduce controlled stochasticity through the Beta distribution.

Beta Distribution Properties. The Beta distribution is a continuous probability distribution defined on the interval $[0, 1]$, making it particularly well suited for modeling proportions and probabilities. Its flexible shape is controlled by two parameters α and β. Under the parameterization $\alpha = r\kappa$ and $\beta = (1-r)\kappa$, the distribution has mean

$$\mathbb{E} = \frac{\alpha}{\alpha + \beta} = r$$

and the concentration parameter κ controls the shape of the distribution.

Stochastic Sampling Phase. We model each sample's error probability using a Beta distribution parameterized by the target error rate r and concentration parameter κ. For each sample i, we independently draw an error probability

$$q_i \sim \text{Beta}(r\kappa, (1-r)\kappa).$$

The parameterization ensures that the expected per-sample error probability satisfies $\mathbb{E}[q_i] = r$, so that the expected global error rate converges to the target value r. This stochastic sampling introduces natural heterogeneity across samples, reflecting realistic scenarios in which data quality varies at the instance level. In our experiments, we set $\kappa = 10$, which provides moderate variance and produces the relationship between atomic-level error rates and the proportion of corrupted samples shown in Figure 19.

Normalization, Adjustment, and Quantization. To ensure that the expected total number of errors matches the target, we first compute the aggregate statistic

$$S = \sum_{j=1}^{M} n_j q_j,$$

and derive the normalization constant as

$$c = \frac{r \cdot N}{S}.$$

The adjusted error probability for sample i is then given by :

$$p_i = \text{clip}(c \cdot q_i, 0, 1).$$

This correction enforces $\sum_{i=1}^{M} n_i p_i \approx r \cdot N$ while maintaining the sample-level heterogeneity introduced by the Beta sampling. The clipping operation ensures all probabilities remain within $[0, 1]$.

Since error counts must be integers, we set the error count for each sample to

$$e_i = \text{round}(n_i \cdot p_i),$$

which rounds $n_i \cdot p_i$ to the nearest integer, and compute the residual $\Delta = \text{round}(r \cdot N) - \sum_{i=1}^{M} e_i$. To correct this discretization error, we iteratively adjust error assignments to ensure that the final assignment satisfies

$$\sum_{i=1}^{M} e_i = \text{round}(r \cdot N).$$

The complete procedure is detailed in Algorithm 2, where $\text{sign}(x)$ denotes the sign function that returns $+1$ if $x > 0$, -1 if $x < 0$, and 0 if $x = 0$.

Algorithm 2: Beta Distribution Error Assignment

Input : atomic function counts $\mathbf{n} = \{n_1, \ldots, n_M\} \in \mathbb{N}^M$
Input : target error rate $r \in [0, 1]$, concentration κ (default 10)
Output : error counts $\mathbf{e} \in \mathbb{N}^M$ with $\sum_i e_i = T$

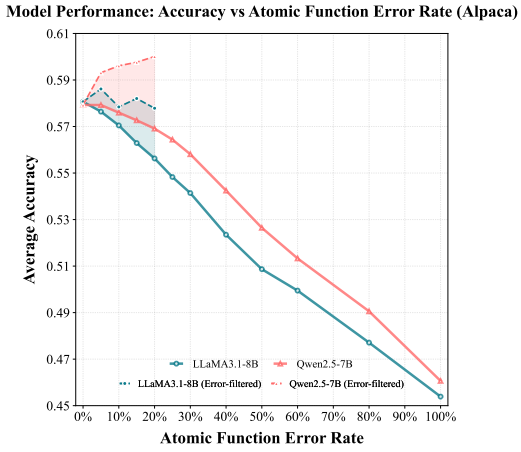
$N \leftarrow \sum_i n_i; \quad T \leftarrow \text{round}(rN)$
`// Sample per-sample error probabilities`
forall $i \in \{1, \ldots, M\}$ **do**
  $q_i \sim \text{Beta}(r\kappa,\ (1-r)\kappa)$
`// Normalize so expected total matches` T
$c \leftarrow rN \,/\, \sum_j n_j q_j$
forall $i \in \{1, \ldots, M\}$ **do**
  $p_i \leftarrow \text{clip}(c\, q_i,\ 0,\ 1)$
  $e_i \leftarrow \text{round}(n_i\, p_i)$
`// Correct rounding residual`
$\Delta \leftarrow T - \sum_i e_i; \quad s \leftarrow \text{sign}(\Delta)$
$\sigma \leftarrow \text{argsort}(-s \cdot \mathbf{n})$; `// large-`n_i `first if` $\Delta > 0$
for $i \in \sigma$ **do**
  if $\Delta = 0$ **then break**;
  if $0 \le e_i + s \le n_i$ **then**
    $e_i \leftarrow e_i + s$
    $\Delta \leftarrow \Delta - s$
return $\mathbf{e}$

F Prompts

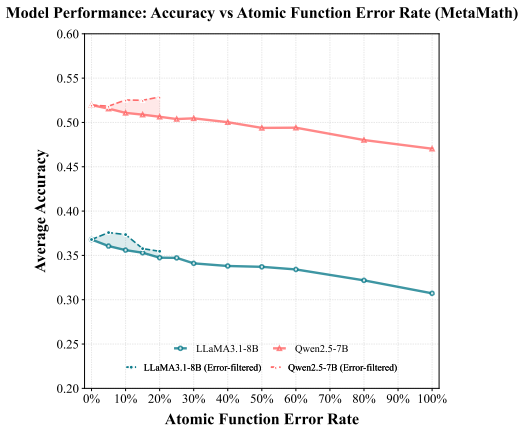
This section enumerates all prompts utilized in our framework. Prompts 1, 2, 3, and 4 detail the construction process of the Atomic Tree. Subsequently, Prompts 5 and 6 outline the creation of Atomic-QA, while Prompts 7, 8, and 9 correspond to Atomic-X pipeline in §7.

G Examples of Atomic Function

In this section, we present examples of atomic functions. Figures 20, 21, and 22 illustrate instances from the Atomic-QA, Alpaca, and Metamath datasets, respectively. In each figure, the leftmost panel displays the original input prompt and its corresponding response. The central columns list the extracted atomic functions followed by the perturbed atomic functions. Finally, the rightmost panel shows the output response after perturbation.



(a) Alpaca accuracy degradation under the perspective of erroneous atomic functions ratio.



(b) MetaMath accuracy degradation under the perspective of erroneous atomic functions ratio.

Figure 18: Accuracy degradation under increasing atomic function error rates for (a) LLaMA and (b) Qwen. The x-axis denotes the proportion of erroneous atomic functions in the training set, while the y-axis represents the performance of the trained model.

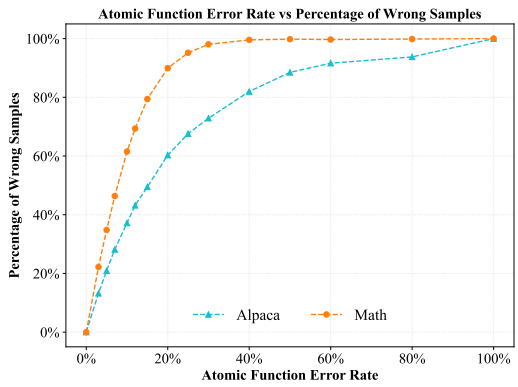


Figure 19: Relationship between the atomic function error rate and the resulting percentage of wrong samples.

Prompt 1: Atomic Function Extraction

```
You are the ENCODER agent. Your job is to extract atomic functions from the Sample Output and
represent each input as (operand, operator, output) triples.

{atomic function definition}

Return **exactly one JSON array** of atomic functions. If no atomic functions can be extracted
, return [].

## Atomic Function Definition
{atomic function definition}

## Rules
1) Minimality: one predicate per triple. Do not merge multiple assertions. If the current
sample is already the smallest unit satisfying the definition, no splitting is required.
2) Order: emit triples in the order their evidence appears in the Sample Output.
3) Numbers/dates: keep literal values and units as written (e.g., "18", "2019", "3.14", "USD")
.
4) Determinism: no randomness; if uncertain, omit the item.
5) Strict output: JSON array only, no comments, no trailing commas, no extra keys.

## Sample Input
This section provides the original input of the data sample. Such as a question, instruction,
or code comment.
{sample input}

## Sample Output
This section provides the original answer of the data sample. It may contain multiple
sentences, equations, or code snippets.
{sample output}

## Output Format
Your answer should wrapped in a proper Json code block using triple backticks like this:
```json
{example}
```
```

Prompt 2: Initial Tree Generation

```
The following are atomic functions, followed by their definitions.

Input atomic functions:
{atomic functions}
The definitions of the atomic functions:
{definitions of atomic functions}

Requirements:
1. Format the output as a JSON tree structure.
2. Consider both coarse-grained and fine-grained classification of atomic functions'
categories, building parent-child relationships.
3. Each node is a dict with category names as keys.
4. The root node should be {root name}.
5. Leaf nodes should contain empty dicts.
6. Wrap the JSON with <begin> and <end>.

Think step by step:
1. Identify the categories of the atomic functions
2. Organize the categories (not atomic functions) into a hierarchical tree structure.

Output format:
<think>
[your reasoning process]
</think>
<begin>
{example}
<end>
```

Prompt 3: Tree Duplication

```
Please perform strict deduplication on the following subcategories under the parent category
"{parent category}".

Requirements:
1. Remove any semantic duplicates (even if names are different)
2. Ensure all items strictly belong to "{parent category}", remove any unrelated items
3. Eliminate any hierarchical relationships between subcategories, remove subcategories that
are included in other subcategories
4. Keep the most appropriate names
5. Output the cleaned list

Current subcategories:
{subcategories}

Output format:
```json
{{"unique_categories": ["example1", "example2"]}}
```
```

Prompt 4: Tree Expansion

```
As a taxonomy expert, suggest 3-5 possible {width/depth} expansion candidates for category: [{
category}]

Related Nodes:
- Parent: {parent}
- Siblings: {siblings}

Definitions:
- Current Node: {definition}
- Parent: {parent_definition}
- Siblings: {siblings_definitions}

Guidelines:
1. Ensure logical consistency with parent/sibling definitions
2. Avoid overlaps with existing nodes
3. Use clear and specific naming

Output JSON format:
```json
{{"new_nodes": ["name1", "name2"]}}
```
```

Prompt 5: Operand Generation

```
You are given the following information:
    - Category
    - Operator
    - Existing Operand

Your task is to generate new operands that are valid inputs for the given operator under the
specified category.
Your generation should consider the following aspects:
1. The generated operands should be specific and fall under the given category.
2. The generated operands should be valid inputs for the given operator.
3. The generated operands should not be similar to the existing operands.
4. Each generated operand should have a brief description that help understand the operands.

## Category ##
This section contains the category that you need to generate operands for.
- Category Name: {category}
- Category Definition: {definition}

## Operator ##
- Operator: {operator}
- Operator Definition: {operator_definition}

## Existing operands ##
This section contains the existing operands that are already classified under the category.
You should avoid generating operands that are similar to the existing ones.
- Existing operands:
{existing operands}

## Output format ##
You should output the generated operands in the JSON format, wrapped in a proper Json code
block using triple backticks like this:
```json
{{
    "operand_name1": "Description of operand 1",
    "operand_name2": "Description of operands 2",
    ...
}}
```
```

Prompt 6: Verification

```
You are given the following information:
    - Function: {function}
    - Operand: {operand}
    - Output: {output}

Your task is to verify whether the function is logically consistent, that is, whether **
Function(Operand)** equals the **Output**.

## Output format ##
You should output the judgement in the JSON format, wrapped in a proper Json code block using
triple backticks like this:
```json
{{
    "judgement": "pass"/"fail",
    "reason": "reason for your judgement"
}}
```
```

Prompt 7: Encoder

```
You are the ENCODER agent. Your job is to extract atomic functions from the Sample Output and
represent each atomic function as a (subject, relation, object) triple.

## Categories
This chapter enumerates the possible candidate categories for decomposing atomic functions.
You need to determine whether the **## Payload** can be further decomposed within some
category.
{categories}

## Sample Input
This section provides the original input of the data sample. Such as a question, instruction,
or code comment.
{sample input}

## Sample Output
This section provides the original answer of the data sample. It may contain multiple
sentences, equations, or code snippets.
{sample output}

## Payload
This section provides the original answer of the data sample. It may contain multiple
sentences, equations, or code snippets.
{sample output}

## Output Format
Your answer should wrapped in a proper Json code block using triple backticks like this:
```json
{example}
```
```

Prompt 8: Perturber

```
You are the PERTURBER agent. Your job is to apply controlled edits to specific atomic
functions in the Sample Output, guided by the provided (subject, relation, object) triple and
payload location.

A *controlled edit* is a minimal, deterministic change that makes the function incorrect while
 preserving overall coherence and fluency.

Return **exactly one JSON object** describing the edit.

Use the Sample Input only to preserve coherence (e.g., pronouns); all edits **must** be
grounded in the Sample Output and the given function.


## Sample Input
This section provides the original input of the data sample (e.g., question/instruction/code
comment).
{input_sample}

## Sample Output
This section provides the original answer of the data sample (sentences, equations, or code).
{output_sample}

## Atomic function to be Perturbed (from ENCODER)
This section provides the function to be perturbed, along with its unique fid and the
corresponding payload location in the Sample Output.
function: {triple}
Payload: {payload}

## Output Format
Your answer should wrapped in a proper Json code block using triple backticks like this:
## Output Format
Your answer should wrapped in a proper Json code block using triple backticks like this:
```json
{example}
```
```

Prompt 9: Decoder

```
You are the DECODER agent. Your job is to reconstruct an output sample from the perturbed
atomic function that is provided.

You shoudld use the SEARCH-REPLACE strategy to ensure the perturbed function is correctly
integrated into the Sample Output while maintaining overall coherence and fluency.

You are allowed to use the following tools, and you must use one tool at a time:
- SEARCH-REPLACE: Locate the content in the Sample Output and replace it with the given
content.
For each change, follow this format exactly:
```[TOOL_CALL]
>>> SEARCH
<original snippet copied from the Sample Output>
======
>>> REPLACE
<modified snippet you want to replace the original snippet with>
======
```

- TERMINATE: Indicate the end of the editing process and output the final revised Sample
Output.
To finish the editing process, follow this format exactly:
```[TOOL_CALL]
Terminate(reason="The reason for termination, e.g., no more changes needed")
```

## Sample Input
This section provides the original input of the data sample (e.g., question/instruction/code
comment).
{input_sample}

## Sample Output
This section provides the original answer of the data sample (sentences, equations, or code).
{output_sample}

## Perturbed Atomic function (from PERTURBER)
This section provides the perturbed function along with the type of edit applied. You should
use change the corresponding content that matches the Original function in the Sample Output
to the Perturbed function.
Original function: {pre_triple}
Perturbed function: {post_triple}

## Output Format
IMPORTANT: Only use one tool call per response. Do not use multiple tool calls in a single
response or code block.
IMPORTANT: You ONLY need to edit the sepcific function that is perturbed. Do NOT make any
other changes to the Sample Output. Do NOT consider the effect of edited function on other
parts of the Sample Output.
Your tool calls should be wrapped in a proper [TOOL_CALL] code block using triple backticks
like this:
```[TOOL_CALL]
<your tool call here>
```
```

Figure 20: The example of Atomic-QA.

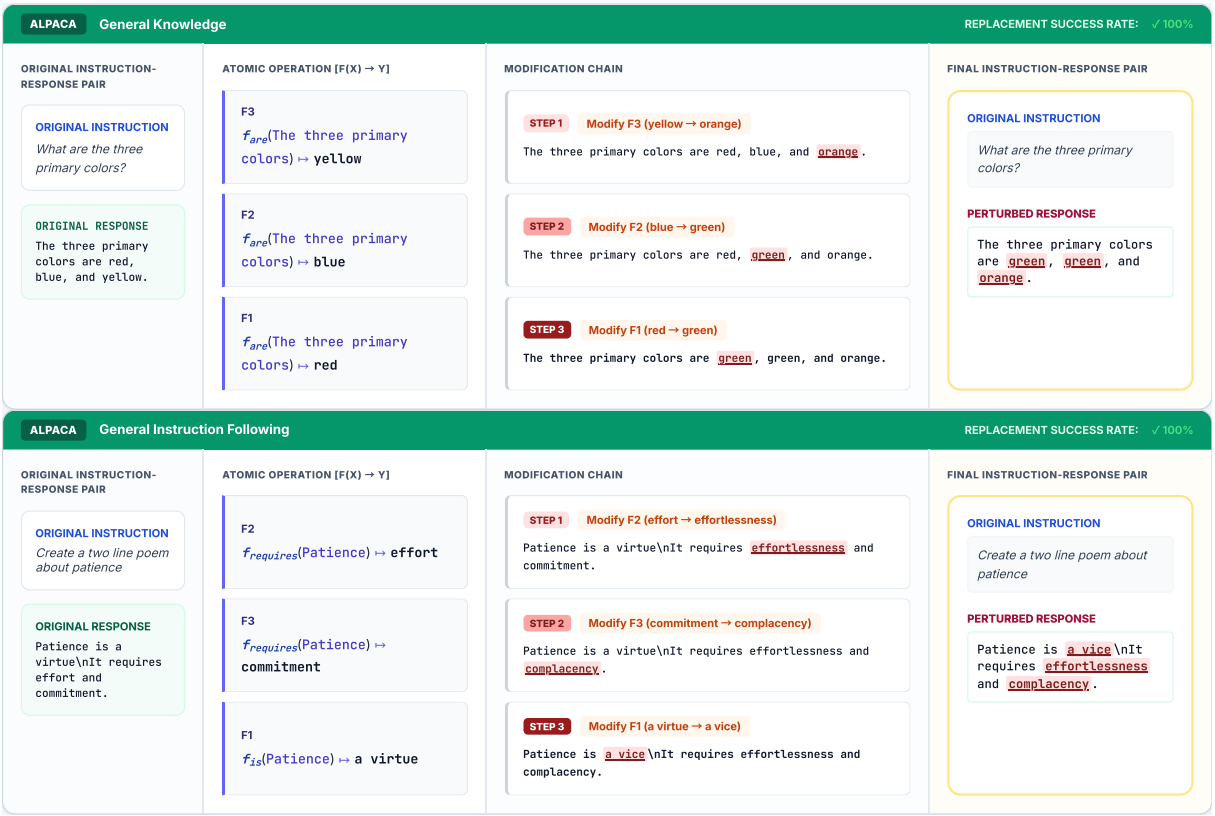


Figure 21: The example of Alpaca.

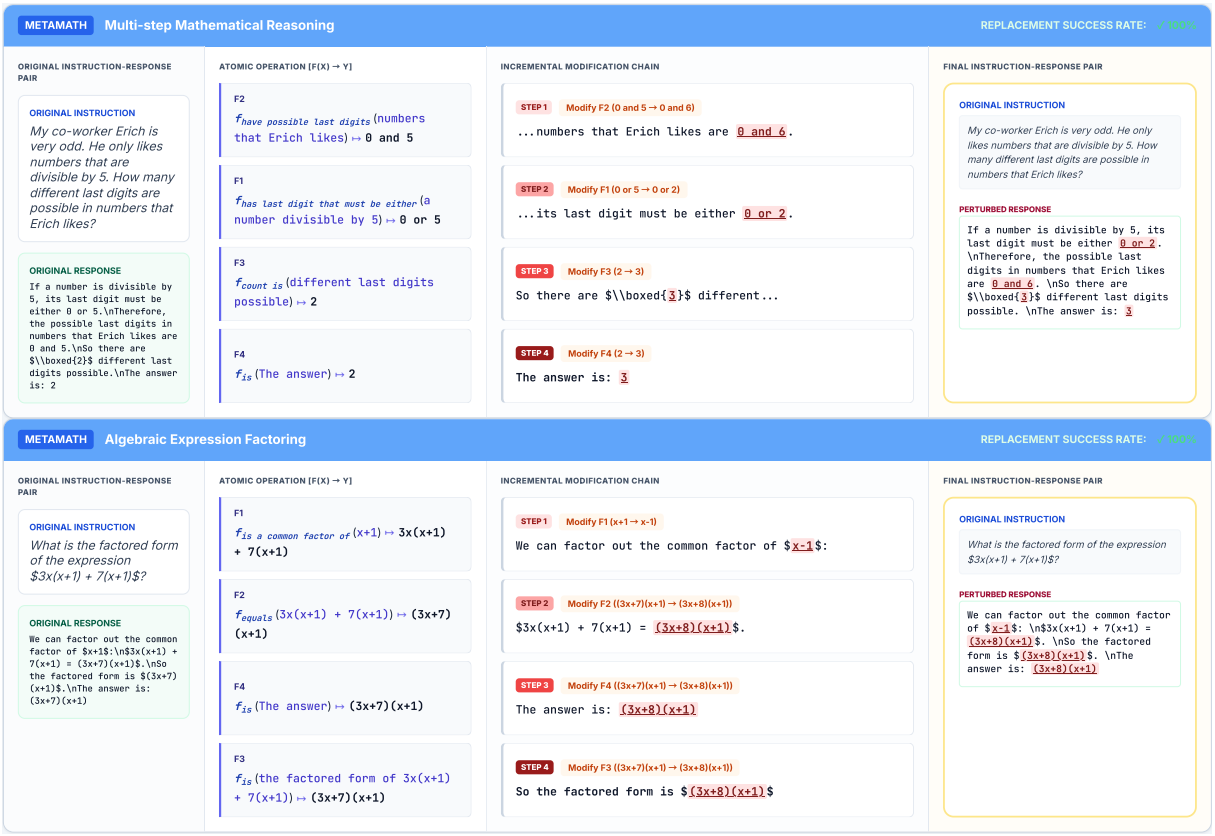


Figure 22: The example of MetaMath.