Lecture 14 — Data II: Filtering, Deduplication, Mixing, and Post-Training Data

Language Modeling from Scratch · lecture notes · rebuilt for NVIDIA Blackwell, late 2026

Section 1: The End-to-End Data Pipeline and Lossy Transformation

Raw web crawls, archival dumps, and source-code repositories cannot be fed directly into a language model. Recall from Lecture 13 that collecting raw data involves moving from live services such as GitHub through crawls and dumps into raw archival datasets while navigating licensing and fair use. Those raw archives arrive as heterogeneous, noisy byte streams packed with markup, visual layout instructions, boilerplate menus, truncated binary files, and massive duplication. At late-2026 frontier pretraining scale —for example, **DeepSeek-V4.1-Flash** (released September 2026) was pretrained on **45 trillion (45T) tokens**, compared to earlier baselines such as **Llama 3** (July 2024) at **15T tokens** and **Qwen3** (May 2025) at **36T tokens**—every stage that converts raw bytes into training tokens must both preserve linguistic signal and scale cleanly across tens of trillions of tokens.

The end-to-end language model data pipeline consists of six stages arranged sequentially from left to right. On the far left sits the **Crawl / Dump** stage covered in Lecture 13. An arrow leads from that raw archive into a four-stage **Pretraining Data Pipeline**:

1. **Transform**: convert rich structured formats such as HTML and PDF into linear text sequences.
2. **Filter**: retain desirable documents and discard noise using language identification, quality scoring, and toxicity filtering.
3. **Dedup**: remove exact and near-duplicate content across the corpus.
4. **Mix**: assign sampling weights across distinct data sources to construct the training distribution.

From the Mix stage, a final arrow leads into stage 5, **Post-Training**, which encompasses mid-training, supervised fine-tuning (SFT), and synthetic data generation tailored closely to target evaluations.

Before any filtering, deduplication, or mixing can begin, we confront the first bottleneck: language models consume one-dimensional token sequences, whereas raw sources arrive in rich, multi-dimensional structured formats.

> **DEFINITION**
> **Linearization** is the process of converting a multi-dimensional or structured document format—such as an HTML DOM tree, a multi-column PDF page, or a hierarchical code repository—into a single one-dimensional sequence of plain-text tokens suitable for language model training.

Every transformation from a structured format to plain text is **inherently lossy**. Two-dimensional spatial layout, embedded images, complex tables, inline mathematical typesetting, and directory hierarchies cannot be represented natively in a flat token stream without losing information or introducing structural

ambiguity. Three primary source formats dominate pretraining corpora, each presenting distinct failure modes during linearization:

Source Format	Typical Origin	What Gets Lost / Hardest Challenge
HTML (main volume)	Common Crawl web pages	Boilerplate (navigation bars, ads, footers), tables, inline math, scripts
PDF	arXiv, web documents, books	Multi-column layout, equations, figures, truncated crawls
Code Repositories	GitHub dumps / archives	Directory hierarchy, cross-file dependencies, commit history

Empirical studies—most notably DataComp for Language Models (**DCLM**)—confirm a "garbage in, garbage out" law at pretraining scale: the quality of text extraction measurably shifts downstream model accuracy even when starting from the exact same raw web crawl.

HTML-to-Text Extraction

Because web pages from Common Crawl supply the largest volume of raw pretraining data, HTML-to-text extraction is the single highest-throughput transformation stage. Picture the structure of a typical raw HTML web page laid out as a vertical wireframe:

- Across the top sits a red **Nav / Header** banner containing site menus and search bars.
- In the middle sits the green **Main Article** column containing the true prose, possibly interleaved with tables and code blocks; immediately to its right runs a narrow red **Ads** sidebar.
- Underneath the article sits a red **Cookie / Footer** strip with legal disclaimers and copyright links.
- At the bottom sits an ambiguous yellow **Images / Captions?** block whose alt-text or captions may or may not carry useful context.

An HTML parser maps this structured wireframe into a clean **Extracted Text** block consisting exclusively of the coherent paragraphs from the main article, stripping away the red boilerplate regions while attempting to preserve the semantic content of tables, code blocks, mathematical markup, and informative captions.

> **DEFINITION**
> **Boilerplate** is repetitive, non-article content embedded around a web page's main prose—such as navigation menus, headers, sidebars, advertisements, cookie consent notices, and footers—that adds noise and artificial repetition if left in a training corpus.

At web scale, **rule-based extraction tools** dominate because neural models are far too slow to parse billions of HTML pages. The standard open-source tools are `trafilatura`, `resiliparse`, `jusText`, and `lynx`. Even among mature rule-based extractors, hard edge cases—especially tables, code blocks, mathematical markup (such as MathJax or LaTeX embedded in HTML spans), and image captions—frequently break or vanish during parsing.

Crucially, DCLM established an important empirical finding regarding Common Crawl's file formats: Common Crawl distributes both raw HTML archives (**WARC** files) and pre-extracted plain-text files (**WET** files). Re-extracting text directly from raw Common Crawl WARC files using a modern extractor such as `resiliparse` or `trafilatura` consistently produces stronger downstream language models than training on Common Crawl's pre-extracted WET files.

COMMON MISCONCEPTION

You might think that using Common Crawl's pre-extracted WET plain-text files is equivalent to parsing raw WARC HTML files, merely saving engineering effort and compute. That breaks because WET extraction strips HTML tags crudely without carefully separating main article prose from navigation bars, footers, and sidebars, leaving heavy boilerplate contamination and mangled formatting that measurably degrades downstream model accuracy.

WORKED EXAMPLE

Evaluating HTML Extraction Yield and Lossiness

Consider a single raw Common Crawl HTML page whose visible text contains $1{,}800$ tokens:

- Navigation header and menus: 420 tokens
- Sidebar advertisements and related-link widgets: 310 tokens
- Cookie banner and footer links: 270 tokens
- Main article prose: 700 tokens
- Main article code snippet and inline table: 100 tokens

Step 1: Naive tag-stripping (approximating raw WET output). Stripping HTML tags without boilerplate detection retains all visible text:

$$\text{Total tokens} = 420 + 310 + 270 + 700 + 100 = 1{,}800 \text{ tokens}.$$

The fraction of true article content is:

$$\frac{700 + 100}{1{,}800} = \frac{800}{1{,}800} \approx 44.4\%,$$

meaning 55.6% of the training tokens spent on this document are non-article boilerplate.

Step 2: Boilerplate-aware extraction (e.g., `trafilatura` on WARC). The parser identifies and discards all $420 + 310 + 270 = 1{,}000$ boilerplate tokens. Suppose it preserves all 700 prose tokens cleanly, and retains 80 of the 100 code/table tokens while dropping 20 tokens of broken table markup:

$$\text{Extracted tokens} = 700 + 80 = 780 \text{ tokens}.$$

Interpretation: Boilerplate-aware extraction reduces token volume by $(1{,}800 - 780)/1{,}800 = 56.7\%$ while raising the article-signal purity from 44.4% to 100%. Simultaneously, the loss of 20 code/table tokens illustrates why extraction is inherently lossy: even high-quality parsers struggle on

> structured edge cases embedded inside prose.

PDF Extraction at Scale: FinePDFs

PDFs—originating from arXiv papers, technical reports, web documents, and books—are among the highest-value sources of structured reasoning and technical knowledge, yet they are far harder to linearize than HTML. Unlike HTML, a PDF stores rendering instructions specifying exact (x, y) page coordinates for glyphs and lines rather than a semantic reading order. Two-column academic papers, multi-line mathematical equations, and embedded figures routinely scramble under naive text dumpers.

A second, less obvious failure mode occurs before parsing even begins: the **truncation problem**. Common Crawl caps the maximum download size per URL, so large PDFs stored inside Common Crawl are cut off mid-file and cannot be parsed cleanly without recovering the full binary.

The **FinePDFs** pipeline addresses both problems through a five-stage linear workflow:

1. **Common Crawl**: identify raw PDF links present in web crawls.
2. **Recrawl Full File**: re-download the complete PDF from its original URL to fix truncated files.
3. **OCR / Parsing**: convert pages to structured text using a combination of fast layout parsers (**Docling**) and vision-language optical character recognition (**RolmOCR**), engineered for high throughput.
4. **Cleanup & Filter**: run post-processing heuristics and filters to repair residual extraction artifacts.
5. **Clean Text**: output the finalized documents to the training corpus.

Even with modern vision-language OCR and dedicated layout parsers, heavy post-processing cleanup and filtering remain mandatory, and substantial visual and layout structure is inevitably lost when flattening pages into one-dimensional text.

> **KEY IDEA**
> Transformation from HTML, PDFs, and code repositories into flat token sequences is inherently lossy, and extraction fidelity directly governs downstream model quality: re-extracting HTML from raw WARC files outperforms pre-extracted WET text, while PDF pipelines such as FinePDFs must recrawl truncated files and pair layout parsers (Docling) with vision-language OCR (RolmOCR).

Extraction converts noisy raw bytes into readable plain text, but readable plain text from the open web is still overwhelmingly dominated by spam, machine-generated junk, irrelevant languages, and toxic content. We now need a systematic, scalable mechanism to decide which extracted documents to keep and which to discard.

✎ Check Your Understanding

1. **Yes/No with justification:** If you already have Common Crawl's pre-extracted WET plain-text files on disk, should you skip parsing the raw WARC HTML files because the textual content is identical?
   Answer: No. DCLM empirically showed that re-extracting text from raw WARC HTML files using boilerplate-aware parsers (such as `trafilatura` or `resiliparse`) measurably improves downstream model quality over pre-extracted WET files, which retain substantial navigation, ad, and footer boilerplate.
2. **Fill-in-the-blank:** Because Common Crawl caps per-file download sizes, large PDFs are often cut off mid-file; the FinePDFs pipeline fixes this before running Docling or RolmOCR by adding a _____ stage.
   Answer: Recrawl (specifically, recrawling the full PDF file from its source URL).

Section 2: Filtering as Target Extrapolation and Scoring Functions

Once documents have been transformed into plain text, we face a selection problem over billions of pages. Hand-writing brittle rules to define "good writing" or "mathematical text" across the entire web does not generalize well and misses nuance. Instead, modern data pipelines formulate filtering mathematically as **target extrapolation**.

DEFINITION

Target extrapolation is the general data-filtering paradigm in which we are given a small, clean set of target documents T and a huge, noisy raw pool R, fit a fast scoring function $\text{score}(x)$ using T and R, and select a large subset $T' \subseteq R$ of documents satisfying $\text{score}(x) \geq \tau$ that resemble T.

Visualize this workflow as a four-stage diagram. On the left sit two input pools stacked vertically: a green box for **Target Data** T (small, clean examples) and a larger gray box for **Raw Data** R (a huge, noisy web pool). Arrows from both T and R converge into a central yellow box labeled **Fit** $\text{score}(x)$ **on** (T, R). From there, an arrow runs into a blue decision box labeled **Keep** $x \in R$ **if** $\text{score}(x) \geq \tau$, which outputs a large green box on the right representing $T' \subseteq R$ (large, similar to T).

This single abstraction unifies the three primary applications of pretraining data filtering: **language identification**, **quality filtering** (including domain-specific filtering), and **toxicity filtering**. For any filtering method to succeed in practice, it must satisfy two simultaneous desiderata:

1. **Generalization beyond** T: We want fresh documents $T' \neq T$ that share the desirable properties of T, not mere copies of T (otherwise we gain no new training volume).
2. **Extreme speed**: Because $\text{score}(x)$ must evaluate every single document in R at multi-trillion-token scale, the scorer must run with minimal compute per document.

Generative vs. Discriminative Scoring Functions

Implementing target extrapolation follows a two-step recipe: (1) fit a fast statistical model from T and R to produce $\text{score}(x)$; (2) retain documents $x \in R$ with $\text{score}(x) \geq \tau$, either deterministically or stochastically. Two modeling families are used to define $\text{score}(x)$:

1. **Generative Model of T**:

$$\text{score}(x) = p_T(x)$$

   Here we fit a probability distribution p_T directly over documents in the target set T, without explicitly needing negative examples from R. In practice, p_T is implemented as a fast n-gram language model such as **KenLM** trained on T. Because raw sequence probability $p_T(x) = \prod_{i=1}^{L} p_T(w_i \mid w_{<i})$ shrinks exponentially with document length L, we normalize for length using **perplexity**:

$$\text{PPL}(x) = \left(\prod_{i=1}^{L} p_T(w_i \mid w_{<i}) \right)^{-1/L} = \exp\left(-\frac{1}{L} \sum_{i=1}^{L} \ln p_T(w_i \mid w_{<i}) \right).$$

   Let us decompose this formula from the inside out:

   - w_i is the i-th token in document x of length L, and $w_{<i}$ denotes its preceding context (truncated to the last $n-1$ tokens in an n-gram model).
   - $p_T(w_i \mid w_{<i})$ is the conditional probability assigned by the target model to token w_i. Taking $-\ln p_T(w_i \mid w_{<i})$ gives the per-token negative log-likelihood (surprisal in nats).
   - Averaging over all L tokens yields the mean negative log-likelihood per token, removing length bias.
   - Exponentiating converts the average log-loss back into an effective branching factor: a perplexity of 10 means the model is, on average, as uncertain as if choosing uniformly among 10 tokens at each step. As a conceptual unit, $\text{PPL}(x)$ measures how surprised the target-domain model p_T is by document x; keeping documents with **low perplexity** is equivalent to keeping documents with high length-normalized $p_T(x)$.

2. **Discriminative Classifier**:

$$\text{score}(x) = p(T \mid x)$$

   Here we train a binary classifier with documents from T as positive examples and documents sampled from R as negative examples, directly estimating the posterior probability that x belongs to the target distribution rather than raw noise. The standard workhorses are **fastText** (a lightweight linear classifier over averaged word and subword n-gram embeddings) and linear bag-of-words classifiers.

> **COMMON MISCONCEPTION**
> You might think that model-based quality filtering has always been avoided because it introduces bias, and that hand-crafted heuristic rules remain the preferred way to clean pretraining data. Earlier heuristic-only corpora—specifically **C4**, **Gopher**, **RefinedWeb**, **FineWeb**, and **Dolma**—did indeed deliberately avoid model-based quality filters out of concern for bias. However, empirical results pioneered by **GPT-3**, **LLaMA**, and **DCLM** overturned that view: model-based filtering produces substantially stronger language models and is now standard practice.

Application 1: Language Identification

The first filtering application isolates text written in target language(s)—for example, retaining English while filtering out other languages at web scale. The standard tool is the off-the-shelf **fastText** language identification classifier, which supports **176 languages** at negligible per-document compute. Its training set consists of multilingual reference text drawn from **Wikipedia**, **Tatoeba** (a collection of translated example sentences), and **SETimes** (Southeast European news articles).

In the **Dolma** corpus, for instance, English filtering uses a simple discriminative threshold: keep a page if $p(\text{English} \mid x) \geq 0.5$. Despite fastText's speed and convenience, language-ID classifiers exhibit systematic edge-case failures: they tend to **under-score short pages** (too few lexical cues), **code-heavy pages** (programming syntax dilutes natural-language n-grams), and **non-standard dialects** (whose vocabulary diverges from Wikipedia and news prose). Because these failures are invisible in aggregate summary statistics, practitioners must always inspect borderline examples near the threshold.

Application 2: Domain Filtering via Multi-Stage Cascades (OpenMathText)

When the target domain T is rare on the open web—such as rigorous mathematical writing—running a relatively costly classifier across every document in Common Crawl wastes compute on pages that obviously contain no math. **OpenMathText** solves this by organizing generative, discriminative, and rule-based filters into a **multi-stage cascade**.

The OpenMathText pipeline flows left to right across four boxes:

1. **Common Crawl**: raw web pages enter the pipeline.
2. **Stage 1 — Heuristic Rules**: fast string checks (for example, checking whether the page contains LaTeX commands) eliminate the vast majority of obviously non-mathematical pages.
3. **Stage 2 — KenLM (ProofPile)**: surviving pages are scored by a generative KenLM n-gram model trained on **ProofPile** (a curated mathematics corpus); pages are kept only if their perplexity is $< 15{,}000$.
4. **Stage 3 — fastText Classifier**: a discriminative fastText classifier applies a context-dependent threshold—keeping pages with score $\geq \tau = 0.17$ if the page already carries explicit math tags, and requiring a stricter threshold $\tau = 0.80$ otherwise.

This cascade extracts **14.7B tokens** of mathematical text from general Common Crawl. The downstream payoff is dramatic: **1.4B-parameter models** trained on OpenMathText outperform models trained on $20\times$ **more data** from unfiltered general web mixtures.

WORKED EXAMPLE

Perplexity Scoring and Multi-Stage Cascade Speedup

Suppose we run an OpenMathText-style cascade over $10{,}000{,}000$ raw web pages and evaluate two 4-token snippets under a ProofPile KenLM model.

Part A: Computing KenLM perplexity.

- For a math snippet x_{math}, the KenLM model assigns conditional probabilities $(0.25, 0.20, 0.10, 0.50)$ to its $L = 4$ tokens. Its joint probability is:

$$p_T(x_{\text{math}}) = 0.25 \times 0.20 \times 0.10 \times 0.50 = 0.0025.$$

  Its perplexity is:

$$\text{PPL}(x_{\text{math}}) = (0.0025)^{-1/4} = 400^{1/4} = \sqrt{20} \approx 4.47.$$

  Because $4.47 < 15{,}000$, x_{math} passes Stage 2.
- For an off-domain noisy snippet x_{noise}, the model assigns $(10^{-5}, 10^{-5}, 10^{-4}, 10^{-6})$. Its joint probability is 10^{-20}, giving:

$$\text{PPL}(x_{\text{noise}}) = (10^{-20})^{-1/4} = 10^5 = 100{,}000.$$

  Because $100{,}000 \geq 15{,}000$, x_{noise} is rejected before reaching Stage 3.

Part B: Cascade throughput savings. Suppose Stage 1 (LaTeX heuristic) costs $1\,\mu$s/page and passes 5% of pages ($500{,}000$ pages); Stage 2 (KenLM) costs $50\,\mu$s/page and passes 20% of Stage 1 survivors ($100{,}000$ pages); and Stage 3 (fastText) costs $200\,\mu$s/page.

- **Without a cascade** (running Stage 3 directly on all 10^7 pages):

$$T_{\text{direct}} = 10^7 \times 200\,\mu\text{s} = 2{,}000\,\text{s}.$$

- **With the cascade**:

$$T_{\text{cascade}} = (10^7 \times 1\,\mu\text{s}) + (5 \times 10^5 \times 50\,\mu\text{s}) + (10^5 \times 200\,\mu\text{s}) = 10\,\text{s} + 25\,\text{s} + 20\,\text{s} = 5$$

Interpretation: Ordering cheap, high-recall filters ahead of more expensive statistical scorers yields a $2{,}000/55 \approx 36.4\times$ speedup while reserving fine-grained discriminative decisions (such as OpenMathText's dual $\tau = 0.17$ vs. $\tau = 0.80$ thresholds) for plausible candidates.

KEY IDEA

Filtering is target extrapolation—fitting a fast generative scorer $p_T(x)$ (such as KenLM perplexity) or discriminative scorer $p(T \mid x)$ (such as fastText) to select $T' \subseteq R$—and chaining cheap heuristic rules ahead of KenLM and fastText in a multi-stage cascade enables web-scale domain curation like OpenMathText's 14.7B math tokens.

OpenMathText shows how well target extrapolation works when a clear domain reference like ProofPile exists. What happens when we want to filter for general-purpose *writing quality* or *educational value*—where "good" is harder to define—or remove toxic text without stripping legitimate discourse? And how tight should our threshold τ be?

> **✎ Check Your Understanding**
>
> 1. **Quick computation:** A 3-token document receives conditional token probabilities $(0.1, 0.1, 0.1)$ from a KenLM model trained on T. What is its perplexity, and does a lower or higher perplexity correspond to a higher generative score $\text{score}(x)$?
>
>   **Answer:** $\text{PPL}(x) = (0.1^3)^{-1/3} = 10$. Lower perplexity corresponds to a higher length-normalized probability $p_T(x)$, and therefore a higher generative score.
>
> 2. **What would happen if…** you ran fastText language identification with a strict threshold $p(\text{English}) \geq 0.9$ on a mixed corpus containing technical tutorials with heavy Python code blocks?
>
>   **Answer:** Many high-value English tutorials would be mistakenly discarded because fastText language-ID scores tend to under-score code-heavy pages (as well as short pages and non-standard dialects).

Section 3: Quality Filtering, Toxicity Filtering, and Budget-Dependent Thresholds

General-purpose quality filtering asks a harder question than language identification: what should serve as the positive target set T when we want to select high-quality general text from Common Crawl? Over time, the field progressed through two major eras of constructing T: using curated proxy corpora as positives, and using an LLM judge to label a small seed set directly.

Historical Baselines: GPT-3 and LLaMA Quality Filters

Two landmark historical baselines established the discriminative quality-filtering template:

System (Historical Baseline)	Target Positives T	Negatives R	Selection Rule
GPT-3 (2020)	Wikipedia, WebText2, Books1, Books2	Common Crawl	Linear word-feature classifier + stochastic Pareto sampling
LLaMA / RedPajama (earlier)	Web pages **referenced by** Wikipedia	Common Crawl	Keep documents classified positive

Notice the subtle difference in how T was chosen. Rather than using Wikipedia articles themselves (which have a narrow encyclopedic style), LLaMA and RedPajama used *external web pages referenced by Wikipedia*—capturing a much broader spectrum of journalism, academic pages, and essays that Wikipedia editors deemed authoritative.

Even more instructive is GPT-3's selection rule. Instead of applying a hard deterministic threshold $\text{score}(x) \geq \tau$, GPT-3 used **stochastic Pareto sampling**:

```
def keep_document(score: float) -> bool:
    return np.random.pareto(9) > 1 - score
```

Let us examine the probability law behind this one-line rule. In NumPy, `np.random.pareto(a)` draws a non-negative random variable $Y \geq 0$ from a Lomax (shifted Pareto) distribution with shape parameter a and survival function:

$$\Pr[Y > u] = \frac{1}{(1+u)^a}, \qquad u \geq 0.$$

Setting $a = 9$ and $u = 1 - s$, where $s = \text{score}(x) \in [0, 1]$ is the classifier's predicted quality probability, the probability of retaining document x is:

$$\Pr[\text{keep } x \mid s] = \frac{1}{(2-s)^9}.$$

Let us break this apart term by term:

- $1 - s$ measures how far document x's score falls below a perfect score of 1.
- Adding 1 gives $2 - s \in [1, 2]$.
- Raising the denominator to the 9-th power creates a steep drop-off as s decreases from 1, while the heavy tail of the Pareto distribution guarantees a strictly positive survival probability even for lower-scoring documents.

When `score` is near 1, $1 - \text{score} \approx 0$, so almost every document is kept. Crucially, the heavy-tailed Pareto noise allows some lower-scoring documents to survive, preventing hard collapse onto narrow Wikipedia-like prose and preserving stylistic and topical diversity across the corpus.

> **WORKED EXAMPLE**
>
> **Computing Retention Probabilities Under GPT-3 Stochastic Filtering**
>
> Evaluate $\Pr[\text{keep } x \mid s] = (1 + (1-s))^{-9} = (2-s)^{-9}$ across four classifier scores:
>
> 1. $s = 1.0$ (near-certain high quality): $u = 1 - 1.0 = 0$, so
>
> $$\Pr[\text{keep}] = (1+0)^{-9} = 1.000 \quad (100\%).$$

2. $s = 0.9$ (strong quality): $u = 0.1$, so

$$\Pr[\text{keep}] = (1.1)^{-9} \approx \frac{1}{2.358} \approx 0.424 \quad (42.4\%).$$

3. $s = 0.5$ (borderline): $u = 0.5$, so

$$\Pr[\text{keep}] = (1.5)^{-9} \approx \frac{1}{38.44} \approx 0.026 \quad (2.6\%).$$

4. $s = 0.0$ (lowest score): $u = 1.0$, so

$$\Pr[\text{keep}] = 2^{-9} = \frac{1}{512} \approx 0.00195 \quad (0.20\%).$$

Interpretation: Unlike a hard cutoff at $\tau = 0.8$ (which keeps 100% above 0.8 and 0% below 0.8), Pareto sampling smoothly favors top-scoring pages while still admitting roughly 2.6% of borderline pages and 0.2% of low-scoring pages. That small stochastic tail guards against blind spots in the classifier's training set T.

LLM-Labeled Quality Filtering: phi-1

Proxy corpora like Wikipedia still constrain quality to "resembling Wikipedia." What if we want to filter for a nuanced criterion such as *"educational value for a student learning basic coding concepts"*? Scoring every file in a trillion-token corpus directly with a frontier LLM is prohibitively slow and expensive.

An earlier milestone, **phi-1** (a **1.5B-parameter** model built around the *"textbooks are all you need"* philosophy, combining GPT-3.5/GPT-4 synthetic data with filtered code), introduced a powerful distillation pattern for quality filtering. The pipeline flows top to bottom through four stages:

1. **Raw R (Python subset of The Stack)**: start with the full unfiltered Python code corpus and sample **100K files** for labeling.
2. **LLM Judge (GPT-4) $\rightarrow$ Target T**: prompt GPT-4 to rate each of the 100K files for *"educational value for a student learning basic coding concepts,"* creating a clean labeled target dataset T.
3. **Fast Student Classifier**: distill the expensive LLM judge into a cheap classifier—specifically, a **random forest** trained on pretrained **CodeGen embeddings** of the 100K labeled files.
4. **Filtered Subset $T' \subseteq R$**: run the fast random forest across the entire Python subset of The Stack to extract the high-educational-value subset T'.

DEFINITION

LLM-distilled quality filtering is a two-stage filtering pattern in which an expensive language model labels a small seed sample of raw documents according to a natural-language rubric, and a fast student classifier is trained to mimic those labels across the entire corpus.

The efficiency gains on **HumanEval** for a **1.3B-parameter** language model were striking:

- **Unfiltered Python Stack**: **12.19%** after **96K training steps**.
- **Filtered subset**: **17.68%** after just **36K training steps**—both substantially higher accuracy and $2.67\times$ fewer training steps.

Application 3: Toxicity Filtering (Dolma)

The third core filtering application is **toxicity filtering**: identifying and removing abusive, hateful, or harmful web text before pretraining. In the **Dolma** pipeline, fast classifiers are trained on the **Jigsaw Toxic Comments (2018)** dataset—originally created to improve online discussions—which consists of Wikipedia talk-page comments labeled across six binary categories:

`toxic` · `severe_toxic` · `obscene` · `threat` · `insult` · `identity_hate`

Those classifiers then score documents across Common Crawl. Toxicity filtering introduces an unavoidable **design tension**: aggressive toxicity thresholds risk removing benign discussions of sensitive social or historical topics, clinical medical text, or underrepresented dialects that trigger lexical false positives in models trained on Wikipedia talk pages. Just as with language identification, inspecting borderline filtered documents is essential.

Scale-Dependent Effects of Filtering

Having built fast, accurate quality scorers, a natural question arises: what threshold τ should we choose? Should we always keep only the top 1% or 5% highest-scoring documents?

The answer is **no**—there is **no single optimal filtering threshold**. Consider the relationship between training token budget and downstream model performance plotted on a two-axis graph:

- The horizontal axis shows **Training token budget** on a logarithmic scale, increasing from short runs on the left to massive multi-trillion-token runs on the right.
- The vertical axis shows **Downstream performance** increasing upward.
- Two curves cross at a dashed vertical line marked **crossover budget**:
  - A red curve representing a **Strict filter (less, higher quality)** rises steeply at small token budgets, outperforming the looser filter early on, but then **saturates and flattens out** (even dipping slightly) at larger budgets because the aggressively shrunk data pool runs out of unique tokens and forces excessive repetition (**epoching**).
  - A blue curve representing a **Looser filter (more data)** climbs more slowly at small budgets due to lower average document quality, but continues improving steadily at large budgets and **overtakes** the strict filter past the crossover budget because it supplies fresh, non-repeated tokens.

COMMON MISCONCEPTION

You might think that stricter quality filtering always yields a better language model—if keeping the top 10% of Common Crawl improves a 100B-token training run, then keeping the top 2% should improve a 10T-token training run even more. That breaks because filtering trades off average token quality against total unique token volume. Once a strict filter shrinks the surviving pool well below your training token budget, the model must repeat the same documents over and over (over-epoching), causing performance to saturate while a looser filter with more unique tokens pulls ahead.

We will return to this exact tension between data quality, finite pool size, and epoching in full quantitative detail when we get to **Section 6 (Data Mixing and the Epoching Trap)**.

WORKED EXAMPLE

Jointly Selecting Threshold Stringency and Token Budget

Suppose raw filtered Common Crawl contains 50T tokens of English prose. Compare two classifier thresholds:

- **Strict filter**: retains the top 1% of documents $\implies 50\text{T} \times 0.01 = 500\text{B}$ unique tokens.
- **Looser filter**: retains the top 20% of documents $\implies 50\text{T} \times 0.20 = 10\text{T}$ unique tokens.

Now evaluate both pools under two training budgets:

1. **Short training run ($N_{\text{train}} = 200\text{B}$ tokens)**:
   - Under the strict filter, the model sees $200\text{B}/500\text{B} = 0.4$ epochs (zero repetition) of top-1% text.
   - Under the looser filter, the model sees 0.02 epochs of lower-average-quality top-20% text.
   - **Result**: the strict filter wins cleanly.
2. **Long training run ($N_{\text{train}} = 10\text{T}$ tokens)**:
   - Under the strict filter, the model must cycle through the 500B pool $10\text{T}/500\text{B} = 20$ epochs, saturating and overfitting to repeated text.
   - Under the looser filter, the model trains for exactly $10\text{T}/10\text{T} = 1.0$ epoch of fresh tokens.
   - **Result**: past the crossover budget, the looser filter overtakes the strict filter.

Interpretation: Filtering stringency τ cannot be chosen in isolation; it must be tuned jointly with the total training token budget.

> **KEY IDEA**
> Quality filtering has evolved from curated proxies with stochastic Pareto sampling (GPT-3) to distilling LLM judges into fast student classifiers (phi-1), while toxicity filtering (Dolma on Jigsaw) balances harm reduction against benign false positives; in every case, filtering stringency must be tuned jointly with total training token budget because overly strict thresholds starve long runs of unique tokens.

Even after transforming raw files and filtering for language, quality, and toxicity, our surviving corpus hides a silent efficiency killer: the same high-quality or template-like text often appears thousands of times across different URLs and repositories. Removing those redundancies at multi-trillion-token scale is the job of deduplication.

> **✎ Check Your Understanding**
>
> 1. **What would happen if…** GPT-3 replaced its stochastic Pareto rule `np.random.pareto(9) > 1 - score` with a hard cutoff `score >= 0.95`?
>   **Answer:** It would discard all documents scoring below 0.95, risking collapse onto narrow Wikipedia-like prose and losing stylistic and topical diversity (as well as shrinking the total unique token pool).
> 2. **Yes/No with justification:** In the phi-1 filtering pipeline, was GPT-4 run directly over every Python file in The Stack to construct the filtered dataset T'?
>   **Answer:** No. GPT-4 was used only to label a 100K-file seed sample to define T, which was then used to train a fast random forest classifier on CodeGen embeddings that scored all of The Stack.

Section 4: Exact Deduplication and the Hashing Paradigm

Why does filtering alone fail to prevent massive repetition? Because quality classifiers score each document independently. If a high-scoring book, open-source license, or formulaic web template appears on ten thousand different web pages, a quality filter happily accepts all ten thousand copies.

Duplicates on the web fall into two distinct categories:

- **Exact duplicates**: byte-for-byte identical copies, such as Project Gutenberg mirror sites or unmodified GitHub repository forks.
- **Near duplicates**: documents that differ by only a few tokens, such as MIT licenses with different copyright years and author names, standard terms-of-service notices, formulaic SEO templates, and minor copy-paste edits.

The scale of duplication in uncleaned web text is staggering. In the historical baseline corpus **C4**, a single formulaic wedding-decor blurb—*"…by combining fantastic ideas, interesting arrangements, and follow the current trends in the field of that make you more inspired and give artistic touches. We'd be honored if you*

can apply some or all of these design in your wedding…"—was repeated **61,036 times**.

As demonstrated by **Lee et al.**, deduplication delivers two concrete engineering benefits:

1. **Training efficiency**: the model stops wasting optimization steps and compute re-reading identical boilerplate.
2. **Less memorization**: eliminating repeated passages dramatically reduces verbatim regurgitation at generation time, directly mitigating copyright and privacy risks.

The Deduplication Design Space and the $O(N^2)$ Bottleneck

Any deduplication system is specified by three independent design choices:

1. **What is an item?** A single sentence, a multi-sentence span, a paragraph, or a full document.
2. **How do we define a match?** An exact string match, a shared sub-item, or a sufficiently high fraction of shared sub-items (a similarity threshold).
3. **What action do we take on duplicates?** Remove all copies of a duplicated item, or keep one canonical copy and remove the rest.

Unlike transformation and filtering—which inspect each document independently in $O(N)$ time—deduplication is fundamentally a **relational** operation: every item must be compared against *all other items* in the corpus. Checking every pair among N items naively requires $\binom{N}{2} = N(N-1)/2 = O(N^2)$ comparisons. At web scale, quadratic algorithms are completely infeasible. We therefore require **linear-time (or near-linear distributed) algorithms built on hashing**.

Hash Functions: Speed vs. Collision Resistance

> **DEFINITION**
> A **hash function** h maps an arbitrary-length input item x to a compact fixed-size integer or bit string $h(x)$; a **hash collision** occurs whenever two distinct items $x \neq y$ map to the same hash value, $h(x) = h(y)$.

Not all hash functions are built for the same purpose. In data pipelines, choosing the right family depends on the tradeoff between adversarial collision resistance and raw throughput:

Family	Examples	Collision Resistance	Speed & Typical Use
Cryptographic	SHA-256	Adversarially collision resistant	Slow — security, Bitcoin
Non-cryptographic	DJB2, **MurmurHash**, CityHash	Not adversarially resistant	Very fast — hash tables, data pipelines

In Python, computing a fast 32-bit MurmurHash3 value takes a single call:

```
import mmh3
h = mmh3.hash("hello")   # fast 32-bit MurmurHash3
```

Because deduplication scans trillions of tokens and does not defend against an adversary crafting malicious colliding inputs, throughput dominates: non-cryptographic hashes such as MurmurHash are the standard choice.

Exact Deduplication and the MapReduce Pattern

Exact deduplication chooses the simplest point in the design space: **item** = string, **match** = exact hash match, **action** = keep one item per hash group.

```
import itertools, mmh3

items = ["Hello!", "hello", "hello there", "hello", "hi", "bye"]
hash_items = itertools.groupby(sorted(items, key=mmh3.hash), key=mmh3.hash)
deduped_items = [next(group) for h, group in hash_items]
```

This three-line structure captures the exact **MapReduce** pattern used across distributed clusters:

- **Map**: compute the hash key $k = h(x)$ for every item x independently in parallel, emitting `(k, x)`.
- **Shuffle / Group**: group all pairs sharing the same hash key k (represented in the single-machine snippet by `sorted` + `itertools.groupby`).
- **Reduce**: emit only the first representative `next(group)` from each hash group, discarding all remaining copies.

Exact deduplication is simple, has exact semantics and high precision, and parallelizes trivially in MapReduce. However, it has two significant limitations:

1. **It misses near duplicates**: changing a single character or punctuation mark completely alters the hash value, so `"Hello!"` and `"hello"` land in entirely different buckets and both survive.
2. **Sub-document exact dedup risks breaking coherence**: the historical baseline **C4** applied exact deduplication at the level of **3-sentence spans**, keeping only one copy of any repeated 3-sentence window. Excising a 3-sentence span from the middle of an otherwise unique article stitches together two unrelated paragraphs, breaking document coherence and corrupting long-range context.

> **COMMON MISCONCEPTION**
> You might think that using a cryptographic hash like SHA-256 is always safer for exact deduplication because it prevents collisions, or that exact deduplication at the 3-sentence level is strictly better than document-level deduplication because it catches boilerplate embedded inside longer pages. Both assumptions run into practical pitfalls: cryptographic hashes waste substantial throughput when accidental collisions are already negligible at sufficient bit width without an adversary, and deleting 3-sentence spans mid-document introduces abrupt semantic discontinuities that hurt language modeling.

> **WORKED EXAMPLE**
> **Pairwise Complexity vs. Exact Hash Deduplication Trace**

Part A: Why $O(N^2)$ fails at web scale. Suppose a web corpus contains $N = 10^9$ (one billion) documents. Pairwise comparison requires:

$$\binom{N}{2} = \frac{10^9(10^9 - 1)}{2} \approx 5 \times 10^{17} \text{ comparisons.}$$

Even if a machine executed 10^9 string comparisons every second without pause, finishing all pairs would take 5×10^8 seconds—nearly 15.9 **years**. By contrast, hashing every document once requires only 10^9 hash evaluations ($O(N)$ work), followed by distributed hash-key grouping.

Part B: Tracing exact deduplication on `items`. Start with `items = ["Hello!", "hello", "hello there", "hello", "hi", "bye"]`.

1. Compute `mmh3.hash` for each string. Both occurrences of `"hello"` map to the exact same 32-bit integer $h(\text{"hello"})$, while `"Hello!"`, `"hello there"`, `"hi"`, and `"bye"` each map to distinct hash values.
2. Group items by hash key:
   - Bucket $h(\text{"hello"})$: `["hello", "hello"]`
   - Four singleton buckets: `["Hello!"]`, `["hello there"]`, `["hi"]`, `["bye"]`
3. Take `next(group)` from each bucket: `["Hello!", "hello", "hello there", "hi", "bye"]` (ordered by hash key).

Interpretation: Exact deduplication cleanly removes the identical duplicate `"hello"` in linear-scale MapReduce style, but leaves the near-duplicate `"Hello!"` untouched because a single capitalization and exclamation point produce an unrelated hash.

KEY IDEA

Deduplication prevents wasted compute and verbatim memorization; exact deduplication via fast non-cryptographic hashing (such as MurmurHash) and MapReduce grouping avoids the $O(N^2)$ pairwise bottleneck, but misses near duplicates that differ by even a single token.

Because licenses, templates, and minor copy-paste edits differ by just a few tokens—and because a standard hash function intentionally scrambles similar inputs into unrelated buckets—exact hashing cannot catch near duplicates. We need a mathematically principled similarity metric and a specialized hashing scheme where *similar* documents collide with high probability while *dissimilar* documents do not.

✎ Check Your Understanding

1. **Fill-in-the-blank:** For large-scale deduplication over trillions of tokens, data pipelines prefer _____ hash functions such as MurmurHash over cryptographic hashes like SHA-256 because throughput dominates over adversarial collision resistance.
   Answer: Non-cryptographic.
2. **What would happen if…** you applied C4's 3-sentence exact deduplication rule to a corpus of technical articles that frequently quote a standard 3-sentence safety disclaimer in the middle of each article?
   Answer: Every article after the first would have those three sentences excised from its middle, stitching together the text immediately before and after the disclaimer and potentially breaking document coherence.

Section 5: Near-Duplicate Detection with Jaccard Similarity, MinHash, and LSH

To catch near duplicates such as MIT licenses with different author lines or news articles syndicated with slightly different headers, we first need a quantitative measure of document overlap, and then a linear-time hashing algorithm that approximates it.

Representing Documents as Sets: Jaccard Similarity

We begin by converting each document into a set of sub-items—typically contiguous word or character n-grams, called **shingles**. Two documents that differ by a few edits share almost all of their n-grams. We measure the overlap between two shingle sets A and B using **Jaccard similarity**.

DEFINITION

The **Jaccard similarity** of two non-empty sets A and B is the size of their intersection divided by the size of their union:

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|} \in [0, 1].$$

Picture a Venn diagram of two overlapping circles: a blue circle on the left for **Set** A containing `{"1", "2", "3", "4"}`, and a yellow circle on the right for **Set** B containing `{"1", "2", "3", "5"}`. In the overlapping central lens sit `"1"`, `"2"`, and `"3"` ($|A \cap B| = 3$); `"4"` sits exclusively in A and `"5"` sits

exclusively in B, so the combined union holds five distinct elements ($|A \cup B| = 5$). Thus:

$$\text{Jaccard}(A, B) = \frac{3}{5} = 0.6.$$

In Python, computing Jaccard similarity for a single pair is one line:

```
def compute_jaccard(A, B):
    return len(A & B) / len(A | B)
```

We define A and B as **near duplicates** whenever $\text{Jaccard}(A, B) \geq \tau$ for a chosen threshold τ (for example, $\tau = 0.8$ or $\tau = 0.99$). Yet computing `compute_jaccard(A, B)` across all pairs still costs $O(N^2)$. How do we find all document pairs with $\text{Jaccard}(A, B) \geq \tau$ in linear time?

MinHash: Making Collision Probability Equal Similarity

With standard hash functions in Section 4, we wanted collisions to be as rare as possible unless two strings were byte-for-byte identical. For near-duplicate detection, we invert that goal: we want a hash function h over sets whose **collision probability directly tracks set similarity**. **MinHash** achieves this exact property.

> **DEFINITION**
> **MinHash** is a randomized hash function on sets that applies a random hash function (seeded independently) to every element $x \in S$ and returns the minimum hash value, $h(S) = \min_{x \in S} \text{hash}(x)$, satisfying the exact identity $\Pr[h(A) = h(B)] = \text{Jaccard}(A, B)$.

In Python, one MinHash evaluation with a given integer `seed` is:

```
def minhash(S: set[str], seed: int):
    return min(mmh3.hash(x, seed) for x in S)
```

To see why $\Pr[h(A) = h(B)] = \text{Jaccard}(A, B)$ holds exactly, arrange the elements of $A \cup B$ into a **characteristic matrix** whose columns are sets A and B and whose entries indicate membership (1 if present, 0 if absent):

Item	A	B	If min?
`1`	1	1	**match**
`2`	1	1	**match**
`3`	1	1	**match**
`4`	1	0	differ
`5`	0	1	differ

Proof of the MinHash property:

1. Assuming no hash collisions among the distinct elements of $A \cup B$, a random hash function assigns a uniformly random ordering (permutation) to all elements of $A \cup B$. Every element in $A \cup B$ is therefore equally likely to receive the smallest hash value—i.e., to appear first in the induced permutation.
2. Let $x^* = \arg\min_{x \in A \cup B} \text{hash}(x)$ be the element of $A \cup B$ with the globally smallest hash value.
   - **Case 1 ($x^* \in A \cap B$):** Since x^* belongs to both A and B and has the smallest hash in $A \cup B$, it is simultaneously the minimum element of A and the minimum element of B. Therefore $h(A) = \text{hash}(x^*) = h(B)$ (a **match**).
   - **Case 2 ($x^* \in A \setminus B$):** Since $x^* \in A$, we have $h(A) = \text{hash}(x^*)$. Since $x^* \notin B$, every element in B has a strictly larger hash value than $\text{hash}(x^*)$, so $h(B) > h(A)$ (they **differ**).
   - **Case 3 ($x^* \in B \setminus A$):** Symmetrically, $h(B) = \text{hash}(x^*) < h(A)$ (they **differ**).
3. Consequently, $h(A) = h(B)$ if and only if the uniformly random minimum element $x^* \in A \cup B$ falls inside $A \cap B$. Because there are $|A \cap B|$ favorable elements out of $|A \cup B|$ equally likely elements:

$$\Pr[h(A) = h(B)] = \frac{|A \cap B|}{|A \cup B|} = \text{Jaccard}(A, B).$$

WORKED EXAMPLE

Tracing MinHash on $A = \{1, 2, 3, 4\}$ and $B = \{1, 2, 3, 5\}$

Here $A \cap B = \{1, 2, 3\}$ and $A \cup B = \{1, 2, 3, 4, 5\}$, so $\text{Jaccard}(A, B) = 3/5 = 0.6$.

- **Seed 1** produces hash values $\text{hash}_1 : \{1 \mapsto 810,\ 2 \mapsto 14,\ 3 \mapsto 502,\ 4 \mapsto 299,\ 5 \mapsto 77\}$.
  - $h_1(A) = \min(810, 14, 502, 299) = 14$ (attained at item `2`).
  - $h_1(B) = \min(810, 14, 502, 77) = 14$ (attained at item `2`).
  - Since the global minimum item `2` lies in $A \cap B$, $h_1(A) = h_1(B)$: **collision**.
- **Seed 2** produces hash values $\text{hash}_2 : \{1 \mapsto 420,\ 2 \mapsto 905,\ 3 \mapsto 312,\ 4 \mapsto 19,\ 5 \mapsto 640\}$.
  - $h_2(A) = \min(420, 905, 312, 19) = 19$ (attained at item `4`).
  - $h_2(B) = \min(420, 905, 312, 640) = 312$ (attained at item `3`).
  - Since the global minimum item `4` lies in $A \setminus B$, $h_2(A) \neq h_2(B)$: **no collision**.

Interpretation: Across all $5! = 120$ equally likely permutations of $\{1, 2, 3, 4, 5\}$, exactly $3/5 = 60\%$ place `1`, `2`, or `3` first, causing A and B to collide.

Locality-Sensitive Hashing (LSH): Banding MinHash

A single MinHash function is far too noisy to act as a clean near-duplicate detector. Because $\Pr[\text{collide}] = s$ is a straight diagonal line, a pair of unrelated documents with $s = 0.4$ still produces a 40% **false-match rate**, while a genuine near-duplicate pair with $s = 0.8$ suffers a 20% **miss rate**. What

we actually want is a sharp step-like decision curve: collision probability near 0 when $s < \tau$ and near 1 when $s \geq \tau$.

Locality-Sensitive Hashing (LSH) via banding transforms the linear probability s into a sharp S-shaped curve. We compute $n = b \cdot r$ independent MinHash values for each document (using n distinct seeds) and partition those n hashes into b **bands of** r **hashes each**.

> **DEFINITION**
> **MinHash LSH banding** partitions a document's $n = b \cdot r$ independent MinHash signature into b bands of r hashes each and declares two documents a candidate pair if **all** r **hashes match** within **at least one** of the b bands (an AND-within-band, OR-across-bands composition).

Visualize the banding architecture for $b = 3$ bands of $r = 4$ hashes ($n = 12$ total):

- Across the top sit three blue band boxes: **Band 1** holds (h_1, h_2, h_3, h_4), **Band 2** holds (h_5, h_6, h_7, h_8), and **Band 3** holds $(h_9, h_{10}, h_{11}, h_{12})$.
- Directly beneath each band sits a yellow **ALL** r **match (AND)** gate: two documents match in Band k only if their entire r-tuple of hashes in Band k is identical.
- Arrows from all three AND gates merge into a single green box at the bottom labeled **Candidate Pair if ANY band matches (OR)**.

Exact Derivation of LSH Collision Probability

Let $s = \text{Jaccard}(A, B)$, and suppose all $n = b \cdot r$ MinHash functions are mutually independent. We derive the probability that A and B collide as a candidate pair in four steps:

1. **Single hash match:** By the MinHash property, any one hash h_j matches with probability
$$\Pr[h_j(A) = h_j(B)] = s.$$
2. **Fixed band match (AND across r hashes):** Because the r hashes inside a fixed band are independent, all r hashes match simultaneously with probability
$$p_{\text{band}} = s^r.$$
3. **Fixed band mismatch:** Therefore, that fixed band fails to match with complementary probability
$$\Pr[\text{band } k \text{ does not match}] = 1 - s^r.$$
4. **No band matches (independent across all b bands):** Because the b bands use disjoint, independent hashes, the probability that *none* of the b bands matches is
$$\Pr[\text{no band matches}] = (1 - s^r)^b.$$

Taking the complement (at least one band matches — the OR across bands) gives the **LSH collision probability**:

$$\Pr[A \text{ and } B \text{ collide}] = 1 - (1 - s^r)^b.$$

Look at how the nested operations work together: the innermost exponentiation s^r suppresses low similarities exponentially (pushing small s close to 0), while the outer complement $1 - (1 - p_{\text{band}})^b$ amplifies moderate band-match probabilities rapidly toward 1 as b grows. Together, this AND-OR composition acts as a tunable threshold function on Jaccard similarity s.

```python
def get_prob_collision(sim: float, b: int, r: int) -> float:
    prob_match = sim ** r
    return 1 - (1 - prob_match) ** b
```

WORKED EXAMPLE

Evaluating the LSH Collision Probability

Check $s = 0.8$, $b = 5$ bands, and $r = 10$ hashes per band ($n = 50$ total MinHashes):

1. Single-band match probability:

$$p_{\text{band}} = 0.8^{10} \approx 0.10737.$$

2. Single-band mismatch probability:

$$1 - p_{\text{band}} \approx 1 - 0.10737 = 0.89263.$$

3. Probability none of the 5 bands matches:

$$(0.89263)^5 \approx 0.5671.$$

4. Overall collision probability:

$$\Pr[\text{collision}] = 1 - 0.5671 \approx 0.433.$$

Interpretation: Even at a moderately high similarity $s = 0.8$, requiring 10 simultaneous hash matches per band across only 5 bands catches fewer than half (43.3%) of candidate pairs. Increasing b or decreasing r is needed if we want to recall pairs at $s = 0.8$.

Tuning the S-Curve, Phase Transition, and Web-Scale Execution

Plotting $\Pr[\text{collision}]$ on the vertical axis (0 to 1) against Jaccard similarity s on the horizontal axis (0 to 1) reveals how b and r sculpt the S-curve:

- The gray dashed diagonal line ($b = 1, r = 1$) represents a single MinHash: linear with no threshold behavior.

- The blue curve ($b = 10, r = 10$) stays near 0 until $s \approx 0.6$, then climbs sharply through 0.679 at $s = 0.80$ and 0.986 at $s = 0.90$.
- The red curve ($b = 10, r = 20$) keeps b fixed while doubling r: **increasing r sharpens the transition and shifts the curve to the right**, making it strictly harder to match (at $s = 0.80$, $\Pr[\text{collision}]$ drops from 0.679 down to 0.109).
- The green curve ($b = 20, r = 20$) keeps $r = 20$ fixed while doubling b from 10 to 20: **increasing b shifts the curve to the left**, making it easier to match (at $s = 0.85$, $\Pr[\text{collision}]$ rises from 0.327 to 0.546, and at $s = 0.90$ from 0.726 to 0.925).

s	$b = 10, r = 10$	$b = 10, r = 20$	$b = 20, r = 20$
0.70	0.249	0.008	0.016
0.80	0.679	0.109	0.207
0.85	0.888	0.327	0.546
0.90	0.986	0.726	0.925
0.95	1.000	0.988	1.000

Where does the steep transition of the S-curve occur? The transition centers on the similarity $s = t$ where the expected number of matching bands $b \cdot s^r$ equals 1—that is, where a single band matches with probability $s^r = 1/b$. Solving for s yields the **LSH phase-transition threshold**:

$$t = \left(\frac{1}{b}\right)^{1/r}.$$

At $s = t$, substituting $s^r = 1/b$ into the collision probability formula gives:

$$\Pr[\text{collision at } s = t] = 1 - \left(1 - \frac{1}{b}\right)^b \approx 1 - \frac{1}{e} \approx 0.632,$$

using the standard limit $\lim_{b\to\infty}(1 - 1/b)^b = e^{-1}$. Thus, regardless of the specific values of b and r (for moderate or large b), the curve crosses approximately 63% collision probability right at $t = (1/b)^{1/r}$.

COMMON MISCONCEPTION

You might think that to double the sharpness of the LSH threshold around a fixed similarity t, you can simply double r while keeping b constant. That breaks because $t = (1/b)^{1/r}$ also moves sharply to the right when r increases alone (for example, moving from $b = 10, r = 10$ where $t = 0.1^{1/10} \approx 0.794$ to $b = 10, r = 20$ shifts the threshold all the way up to $t = 0.1^{1/20} \approx 0.891$). To sharpen the S-curve while holding t fixed, you must increase *both* r and b together so that $b \approx t^{-r}$.

WORKED EXAMPLE

Production Web-Scale LSH Setting (Lee et al.)

In the web-scale near-deduplication study by Lee et al., the LSH configuration uses $n = 9{,}000$ MinHashes split into $b = 20$ bands of $r = 450$ hashes each.

Step 1: Compute the phase-transition threshold t.

$$t = \left(\frac{1}{20}\right)^{1/450} = \exp\left(-\frac{\ln 20}{450}\right) \approx \exp\left(-\frac{2.9957}{450}\right) \approx e^{-0.006657} \approx 0.993.$$

Step 2: Check collision probability right at $s = t$. At $s = t$, $p_{\text{band}} = 1/20 = 0.05$, and:

$$\Pr[\text{collision}] = 1 - \left(1 - \frac{1}{20}\right)^{20} = 1 - (0.95)^{20} \approx 1 - 0.3585 = 0.642.$$

Step 3: Check just below and just above t to verify sharpness.

- Just below at $s = 0.98$: $p_{\text{band}} = 0.98^{450} \approx 1.13 \times 10^{-4}$, giving $\Pr[\text{collision}] = 1 - (1 - 1.13 \times 10^{-4})^{20} \approx 0.0023$ (0.23%).
- Just above at $s = 0.996$: $p_{\text{band}} = 0.996^{450} \approx 0.1647$, giving $\Pr[\text{collision}] = 1 - (0.8353)^{20} \approx 0.973$ (97.3%).

Interpretation: Across a narrow similarity window from 0.980 to 0.996, collision probability leaps from 0.23% to 97.3%. This extreme sharpness targets near-identical documents while virtually eliminating false collisions among moderately similar pages.

Finally, how does MinHash + LSH execute in **linear time** without pairwise comparisons? For each document x and each band $k \in \{1, \ldots, b\}$, we hash the band's r-tuple $(h_{(k-1)r+1}(x), \ldots, h_{kr}(x))$ into a single compact `bucket_key`, and emit the key-value pair `((band_id, bucket_key), doc_id)`. Grouping by `(band_id, bucket_key)` in MapReduce places any two documents that match on all r hashes of band k into the exact same bucket—requiring only $O(N \cdot b)$ emissions rather than $O(N^2)$ pairwise checks.

KEY IDEA

MinHash converts Jaccard similarity into exact hash-collision probability, $\Pr[h(A) = h(B)] = \text{Jaccard}(A, B)$, and LSH banding ($b$ bands of r hashes) sharpens that linear probability into an S-curve $\Pr[\text{collide}] = 1 - (1 - s^r)^b$ centered at $t = (1/b)^{1/r}$ that executes in linear time via MapReduce grouping on `(band_id, bucket_key)`.

We now have clean, filtered, deduplicated document pools from many distinct origins—Common Crawl, Wikipedia, GitHub, scientific papers, books, and specialized math dumps. Next, we must decide how much training compute to allocate to each source.

✎ Check Your Understanding

1. **Quick computation:** Two shingle sets A and B have $\text{Jaccard}(A, B) = 0.5$. Under MinHash LSH with $b = 2$ bands and $r = 2$ hashes per band, what is the probability that A and B collide as a candidate pair?
   Answer: Each band matches with probability $p_{\text{band}} = 0.5^2 = 0.25$. The collision probability is $1 - (1 - 0.25)^2 = 1 - 0.5625 = 0.4375$.
2. **What would happen if…** you increase the number of bands b while keeping the number of hashes per band r fixed?
   Answer: The LSH S-curve shifts to the left (lower threshold $t = (1/b)^{1/r}$), making it easier for pairs to collide—which increases recall of near duplicates at the cost of more false-positive candidate pairs.

Section 6: Data Mixing, the Epoching Trap, and Controlling Repetition

Clean, deduplicated corpora are never drawn from a single homogeneous source. Real pretraining corpora span dozens of distinct sources—for example, historical baselines like **The Pile** and open corpora like **Marin** combine web crawls, encyclopedias, code repositories, academic papers, and books. Visually, a **data mixture** is a probability distribution $p(s)$ over sources s summing to 1. For instance, a simple three-source mixture might allocate horizontal bars of **Common Crawl:** 0.50, **Wikipedia:** 0.30, and **GitHub:** 0.20.

> **DEFINITION**
> A **data mixture** $p(s)$ is a probability distribution over the distinct data sources in a pretraining corpus that specifies what fraction of training tokens is sampled from each source s.

Three simple baseline strategies are commonly used to set $p(s)$:

- **Vibes / manual tuning**: hand-tuned weights guided by intuition and ad-hoc experiments (still very common in practice).
- **Uniform weighting**: $p(s) \propto 1$, giving every source equal weight regardless of size.
- **Proportional weighting**: $p(s) \propto \text{tokens}(s)$, sampling each source strictly in proportion to its raw token count.

Why not simply assign nearly all sampling weight to our highest-quality sources, such as Wikipedia, curated books, or peer-reviewed papers? Two fundamental constraints prevent that:

1. **Diversity across incomparable domains**: no single source teaches every capability. Literature, source code, scientific papers, and conversational web text cover disjoint skills that cannot substitute for one

another.

2. **Finite pool sizes**: every high-quality source is finite. Giving a small reservoir a large sampling weight forces the training run to cycle through the same documents dozens of times (**epoching**).

The Epoching Trap: Finite High-Quality Reservoirs

Consider a concrete two-pool scenario illustrating how an innocent-looking mixture triggers catastrophic repetition. Picture a flow diagram with two source reservoirs on the left feeding into a central **Training Run** box of $N_{\text{train}} = 1\text{T}$ **tokens**, producing two epoch-count boxes on the right:

- Top left (blue): an abundant **"low"** pool containing 10T **tokens**, assigned $p(\text{low}) = 0.5$, from which 500B tokens are drawn. On the top right (green), its epoch count is:

$$\text{epochs}(\text{low}) = \frac{0.5\text{T}}{10\text{T}} = 0.05 \text{ epochs}.$$

- Bottom left (red): a scarce **"high"** pool containing only 10B **tokens**, also assigned $p(\text{high}) = 0.5$, from which 500B tokens are drawn. On the bottom right (red warning), its epoch count is:

$$\text{epochs}(\text{high}) = \frac{0.5\text{T}}{10\text{B}} = 50 \times \text{ epochs! (overfit)}.$$

A balanced-looking $50/50$ mixture quietly repeats the scarce 10B-token pool **50 times** while barely touching 5% of the abundant pool. At late-2026 frontier pretraining budgets—such as **45T tokens** for **DeepSeek-V4.1-Flash**—finite high-quality reservoirs hit epoch limits even sooner ($45\times$ faster than a 1T run).

> **COMMON MISCONCEPTION**
> You might think that a $50/50$ sampling mixture gives "equal exposure" to two sources without harming either one. That breaks whenever source sizes differ by orders of magnitude: the number of times each token in source s is repeated equals $\frac{p(s)\cdot N_{\text{train}}}{\text{tokens}(s)}$, so a 50% weight on a 10B-token pool during a 1T run forces 50 full passes and severe overfitting.

Controlling Epochs: Power-Law Scaling and UniMax

Two analytical methods prevent small sources from being either drowned out (under proportional sampling) or over-epoched (under uniform sampling):

1. **Power-law (temperature) smoothing** (a classic multilingual balancing heuristic):

$$p(s) \propto \text{tokens}(s)^{\alpha}, \qquad \alpha \in [0, 1].$$

Explicitly, normalizing across all sources s':

$$p(s) = \frac{\text{tokens}(s)^{\alpha}}{\sum_{s'} \text{tokens}(s')^{\alpha}}.$$

- Setting $\alpha = 0$ yields **uniform** sampling ($p(s) \propto 1$), completely ignoring pool size.
- Setting $\alpha = 1$ yields **proportional** sampling ($p(s) \propto \text{tokens}(s)$), keeping every source at the exact same epoch count but starving small high-value sources.
- Choosing an intermediate $\alpha \in (0, 1)$ (such as $\alpha = 0.5$) upweights smaller sources relative to proportional sampling while still respecting their finite token counts.

2. **UniMax** (originally designed for multilingual balancing): Power-law smoothing softens repetition, but does not enforce a strict upper bound on how many times any individual source can repeat. **UniMax** solves this directly by sampling sources **as uniformly as possible** subject to a **hard cap C on epochs** for every source s:

$$\frac{p(s) \cdot N_{\text{train}}}{\text{tokens}(s)} \leq C \qquad \forall s.$$

Let us decompose this inequality term by term:

- $p(s) \cdot N_{\text{train}}$ is the total number of training tokens drawn from source s over the entire training run.
- Dividing by $\text{tokens}(s)$ yields the exact number of epochs (passes) made over source s.
- The inequality forces every source to stay at or below C passes.

Operationally, UniMax executes as an iterative water-filling allocation: attempt to split the remaining training token budget uniformly across all uncapped sources; whenever a small source would exceed $C \cdot \text{tokens}(s)$ tokens, clamp its allocation to exactly C epochs and redistribute its leftover token budget equally across the remaining uncapped sources until every source satisfies the cap.

DEFINITION

UniMax is a data-mixing algorithm that allocates training tokens as uniformly as possible across sources subject to a hard maximum epoch cap C on every source, redistributing leftover budget from capped small sources to the remaining uncapped sources.

WORKED EXAMPLE

Comparing Power-Law Smoothing and UniMax Allocation

Part A: Power-law smoothing ($\alpha = 0.5$) on the two-pool epoching trap. Take `low` $= 10\text{T} = 10^{13}$ tokens, `high` $= 10\text{B} = 10^{10}$ tokens, and $N_{\text{train}} = 1\text{T}$.

1. Compute unnormalized weights $\text{tokens}(s)^{0.5}$:

$$w(\text{low}) = (10^{13})^{0.5} \approx 3.1623 \times 10^{6}, \qquad w(\text{high}) = (10^{10})^{0.5} = 10^{5}.$$

2. Normalize by the sum $w(\text{low}) + w(\text{high}) = 32.623 \times 10^{5}$:

$$p(\text{high}) = \frac{10^{5}}{32.623 \times 10^{5}} \approx 0.03065, \qquad p(\text{low}) \approx 0.96935.$$

3. Compute tokens drawn and epochs on `high`:

$$\text{drawn}(\text{high}) = 0.03065 \times 1\text{T} \approx 30.65\text{B} \implies \text{epochs}(\text{high}) = \frac{30.65\text{B}}{10\text{B}} \approx 3.07 \text{ epocl}$$

Compare this against proportional sampling ($\alpha = 1$, which gives $p(\text{high}) \approx 0.001$ and only 0.1 epochs) and $50/50$ sampling (50 epochs): $\alpha = 0.5$ boosts `high` by $30\times$ over proportional sampling while keeping repetition to a manageable 3.07 epochs.

Part B: UniMax iterative allocation with cap $C = 4$ epochs. Suppose we have three sources—$S_1 = 10\text{B}$, $S_2 = 100\text{B}$, and $S_3 = 10\text{T}$—and a training budget $N_{\text{train}} = 1\text{T} = 1{,}000\text{B}$ tokens with hard epoch cap $C = 4$.

- Maximum allowed tokens per source: $S_1 \leq 40\text{B}$, $S_2 \leq 400\text{B}$, $S_3 \leq 40\text{T}$.
- **Round 1**: Split $1{,}000\text{B}$ uniformly among 3 sources $\rightarrow 333.3\text{B}$ each. S_1 exceeds its 40B cap, so clamp $S_1 = 40\text{B}$ ($p(S_1) = 0.04$, exactly 4 epochs).
- **Round 2**: Remaining budget is $1{,}000\text{B} - 40\text{B} = 960\text{B}$, split uniformly between S_2 and $S_3 \rightarrow 480\text{B}$ each. S_2 exceeds its 400B cap, so clamp $S_2 = 400\text{B}$ ($p(S_2) = 0.40$, exactly 4 epochs).
- **Round 3**: Remaining budget is $960\text{B} - 400\text{B} = 560\text{B}$, all assigned to S_3 ($p(S_3) = 0.56$, 0.056 epochs, well below $C = 4$).

Interpretation: Every source respects the hard 4-epoch ceiling ($40 + 400 + 560 = 1{,}000\text{B}$), while small sources still receive maximum permitted representation without overfitting.

KEY IDEA

Because high-quality sources are finite, naive uniform or $50/50$ mixtures trigger catastrophic epoching at trillion-token scale; power-law smoothing ($p(s) \propto \text{tokens}(s)^{\alpha}$) softens this tradeoff, while UniMax enforces a hard epoch cap $\frac{p(s)N_{\text{train}}}{\text{tokens}(s)} \leq C$ and redistributes leftover budget across uncapped sources.

UniMax guarantees that no source suffers runaway repetition, but notice a blind spot: it still treats all uncapped sources **symmetrically**, ignoring which sources actually drive down validation loss or improve downstream capabilities the most. Can we optimize mixture weights directly from empirical performance data?

✎ Check Your Understanding

1. **Quick computation:** A corpus has a scarce source of 20B tokens assigned mixture weight $p(s) = 0.10$ in a 4T-token training run. How many epochs will the model train on this source?
   Answer: $\text{epochs} = \frac{0.10 \times 4{,}000\text{B}}{20\text{B}} = \frac{400\text{B}}{20\text{B}} = 20$ epochs.
2. **Fill-in-the-blank:** In power-law smoothing $p(s) \propto \text{tokens}(s)^{\alpha}$, setting $\alpha = 0$ produces _____ sampling, whereas setting $\alpha = 1$ produces _____ sampling.
   Answer: uniform; proportional.

Section 7: Automated Data Mixing via RegMix and Simulated Epoching

Rather than relying on manual "vibes" or symmetric rules like UniMax, **Regression-Based Data Mixing (RegMix)** treats the choice of mixture weights p as an empirical optimization problem—directly analogous to scaling laws, where cheap small-scale experiments map out a response surface that is then extrapolated to full training scale.

DEFINITION

RegMix (Regression-Based Data Mixing) is an automated mixing framework that trains many small proxy models on randomly sampled data mixtures $p^{(i)}$, fits a regression surrogate $\hat{L}(p)$ predicting evaluation loss from mixture weights, and selects the optimal mixture $p^* = \arg\min_p \hat{L}(p)$ for the full-scale training run.

The RegMix pipeline flows across five stages from left to right:

1. **Sample Mixtures**: draw candidate probability vectors $p^{(i)} \sim \text{Dirichlet}$ across the simplex of sources.
2. **Small Proxy Runs**: train a small model on each mixture $p^{(i)}$ for a small token budget and measure its target loss or downstream evaluation metric.
3. **Fit Surrogate $\hat{L}(p)$**: fit a regression model—such as linear regression or boosted decision trees—mapping mixture vector $p \mapsto \hat{L}(p)$.
4. **Optimize**: minimize the surrogate over valid probability distributions to find $p^* = \arg\min_p \hat{L}(p)$.
5. **Full Run**: train the target large-scale model using mixture p^*.

A critical caveat applies to Step 2: **be careful with the target metric**. Optimizing p directly against a narrow downstream benchmark risks overfitting the entire pretraining mixture to that specific evaluation suite rather than building general capability.

More fundamentally, RegMix rests on **two critical assumptions ("hopes")**:

- **Hope 1**: the regression surrogate $\hat{L}(p)$ is accurate in the neighborhood of its minimizer p^*.
- **Hope 2**: the optimal mixture p^* discovered at small proxy scale **transfers** faithfully to large target scale.

When Hope 2 Breaks: Scale-Dependent Epoching

Hope 2 fails dramatically whenever finite pool sizes interact with scale—precisely the epoching trap we uncovered in Section 6. Return to our two pools: `low` $= 10\text{T}$ tokens (abundant) and `high` $= 10\text{B}$ tokens (scarce). Suppose a small proxy run trains on $N_{\text{small}} = 10\text{B}$ tokens, while the large target run will train on $N_{\text{large}} = 1\text{T}$ tokens. Compare the exact same aggressive mixture $p = (\text{low}, \text{high}) = (0.1, 0.9)$ at both scales:

Run Scale	Mixture $p = (\text{low}, \text{high})$	Tokens Drawn from `high`	Epochs on `high`	Outcome
Small proxy (10B tokens)	$(0.1,\ 0.9)$	9B	**0.9 epochs**	Looks great (no repetition)
Large target (1T tokens)	$(0.1,\ 0.9)$	900B	**90 epochs**	Severe overfitting!

At the 10B proxy budget, the scarce 10B high-quality pool never runs out of unique tokens, so the proxy model sees 0.9 epochs of pure fresh high-quality text and rewards $p = (0.1, 0.9)$ with an artificially low loss. When we blindly transfer $p^* = (0.1, 0.9)$ to the 1T full run, the model slams straight into the finite size of `high` and repeats it **90 times**.

> **COMMON MISCONCEPTION**
> You might think that because small proxy runs use the exact same underlying data pools as the full run, any mixture that minimizes proxy validation loss will remain optimal when scaled up by $100\times$ in training tokens. That breaks because holding pool sizes constant while shrinking N_{train} destroys epoch equivalence: scarce pools that repeat 90 times at full scale do not even repeat once at proxy scale, causing the proxy optimizer to systematically over-weight small sources.

The Fix: Simulated Epoching

The core principle behind fixing Hope 2 is clean: **make small-scale proxy runs faithfully mirror large-scale dynamics**. Specifically, **simulated epoching** downsamples every source pool by the exact ratio of the small proxy budget to the large target budget:

$$\rho = \frac{N_{\text{small}}}{N_{\text{large}}}.$$

DEFINITION

Simulated epoching is a proxy-experiment technique that downsamples every source pool's size by $\rho = N_{\text{small}}/N_{\text{large}}$ before running small-scale data-mixing experiments so that every candidate mixture experiences the exact same per-source epoch count at proxy scale as it will at full scale.

Verify mathematically why this downsampling makes epoch counts strictly scale-invariant. After scaling every source's available token count to $\text{tokens}_{\text{down}}(s) = \rho \cdot \text{tokens}(s)$, the epoch count experienced by source s during the small proxy run is:

$$\text{epochs}_{\text{small}}(s) = \frac{p(s) \cdot N_{\text{small}}}{\rho \cdot \text{tokens}(s)} = \frac{p(s) \cdot N_{\text{small}}}{(N_{\text{small}}/N_{\text{large}}) \cdot \text{tokens}(s)} = \frac{p(s) \cdot N_{\text{large}}}{\text{tokens}(s)} = \text{epochs}_{\text{large}}(s)$$

The factor N_{small} cancels identically. In Python, implementing the pool-size adjustment takes four lines:

```
ratio = 10e9 / 1e12  # 0.01
downsampled = {
    s: count * ratio
    for s, count in counts.items()
}
```

Picture two stacked comparison boxes illustrating the difference:

- **Top box (red — Naive Small Proxy, 10B run on full pools)**: `low`: 10T, `high`: 10B. Testing $p = (0.1, 0.9)$ gives only 0.9 **epochs** on `high`, hiding overfitting and selecting a bad optimum for the 1T run.
- **Bottom box (green — Simulated Epoching, downsampled by $\rho = 0.01$)**: `low`: 100B, `high`: 100M. Testing $p = (0.1, 0.9)$ immediately forces 90 **epochs** on `high` (matching the 1T run!). Over-epoching now hurts the proxy run itself, steering the surrogate optimizer away from $(0.1, 0.9)$ toward a realistic, balanced large-scale mixture such as $p = (0.7, 0.3)$.

WORKED EXAMPLE

Verifying Simulated Epoching Across Mixtures

Let `low` $= 10\text{T}$ tokens, `high` $= 10\text{B}$ tokens, $N_{\text{small}} = 10\text{B}$, and $N_{\text{large}} = 1\text{T}$.

Step 1: Compute the downsampling ratio and downsampled pools.

$$\rho = \frac{10\text{B}}{1\text{T}} = 0.01.$$

$$\text{tokens}_{\text{down}}(\text{low}) = 0.01 \times 10\text{T} = 100\text{B}, \qquad \text{tokens}_{\text{down}}(\text{high}) = 0.01 \times 10\text{B} = 100\text{M}.$$

Step 2: Evaluate the aggressive mixture $p = (0.1, 0.9)$ on the downsampled proxy.

- Tokens drawn from `high`: $0.9 \times 10\text{B} = 9\text{B}$.
- Proxy epochs on `high`: $9\text{B}/100\text{M} = 90$ epochs—matching the $900\text{B}/10\text{B} = 90$ epochs of the 1T run. The proxy model visibly overfits `high`, penalizing its validation loss.

Step 3: Evaluate the balanced mixture $p = (0.7, 0.3)$ on the downsampled proxy.

- Tokens drawn from `low`: $0.7 \times 10\text{B} = 7\text{B} \implies 7\text{B}/100\text{B} = 0.07$ epochs (matching $700\text{B}/10\text{T} = 0.07$ epochs at 1T).
- Tokens drawn from `high`: $0.3 \times 10\text{B} = 3\text{B} \implies 3\text{B}/100\text{M} = 30$ epochs (matching $300\text{B}/10\text{B} = 30$ epochs at 1T).

Interpretation: Because both pools experience identical repetition dynamics at 10B and 1T, the RegMix surrogate observes the true large-scale quality-versus-overfitting tradeoff and shifts p^* to a mixture that transfers cleanly.

KEY IDEA

RegMix optimizes data mixtures by fitting a surrogate $\hat{L}(p)$ to small proxy runs, but transferring p^* to large scale fails unless **simulated epoching** downsamples every source by $\rho = N_{\text{small}}/N_{\text{large}}$ so proxy runs experience the exact epoch counts of the full run.

That completes all four stages of the pretraining data pipeline—Transform, Filter, Deduplicate, and Mix. Pretraining gives a model broad world knowledge and language fluency, yet raw pretraining text rarely looks like multi-step mathematical proofs, interactive debugging sessions, or agentic repository edits. To unlock frontier reasoning and software engineering capability, we move to stage 5 of our end-to-end pipeline: post-training and synthetic data.

✎ Check Your Understanding

1. **Quick computation:** You plan a 45T-token frontier training run and want to run 45B-token RegMix proxy experiments using simulated epoching. If your Wikipedia pool has 5B tokens, how many tokens should remain in your downsampled Wikipedia pool for the proxy runs?
   Answer: The downsampling ratio is $\rho = 45\text{B}/45\text{T} = 0.001$. Downsampling Wikipedia gives $0.001 \times 5\text{B} = 5\text{M}$ tokens.
2. **Yes/No with justification:** If you optimize RegMix's surrogate $\hat{L}(p)$ directly on a single narrow downstream benchmark, is that guaranteed to improve general pretraining quality?
   Answer: No. Optimizing directly on a narrow downstream evaluation risks overfitting the data mixture to that specific benchmark at the expense of broader capabilities.

Section 8: Post-Training Data, Reasoning Distillation, and Software Engineering Trajectories

Unlike pretraining data—which aims to cover general human knowledge at tens of trillions of tokens—mid-training and post-training data closely resemble target evaluations. Building post-training datasets follows a three-stage workflow flowing left to right:

1. **Define Environments** (blue box): construct the interactive worlds or domains in which tasks live—mathematical proof settings, real code repositories, or execution sandboxes.
2. **Define Tasks / Prompts** (yellow box): populate those environments with prompts that may be **real**, **semi-synthetic**, or **fully synthetic**.
3. **Collect Teacher Data** (green box): generate rollouts and multi-step agent trajectories from strong teacher models and apply filtering.

Late-2026 frontier models combine supervised fine-tuning (SFT), reinforcement learning (RL), and massive multi-teacher distillation in this post-training pipeline:

- **DeepSeek-V4.1-Flash** (September 2026): **SFT** $\rightarrow$ **asynchronous RL** $\rightarrow$ **on-policy distillation from more than 40 teachers**.
- **Nemotron 3.5 Super / Lightning** (October 2026): **SFT** $\rightarrow$ **RLVR (RL with verifiable rewards)** $\rightarrow$ **expert teachers** $\rightarrow$ **MOPD**, trained with a global batch size of **8,192** and **16 rollouts per prompt**.

Reasoning Distillation: OpenThoughts

To see how delicate post-training data decisions are in practice, examine **OpenThoughts**, a systematic study of reasoning distillation that produced **1.2M reasoning examples** distilled using **QwQ-32B** as the teacher over prompts drawn from **27 human and synthetic sources** (including StackExchange, NuminaMath, and Chemistry).

Systematic ablations in OpenThoughts uncovered **four surprising empirical findings** that directly overturn common intuition:

1. **Multiple samples help**: sampling **16 responses per prompt** (rather than a single greedy or single stochastic trace) significantly improves student quality.
2. **Stronger model $\neq$ better teacher**: QwQ-32B outperformed **DeepSeek-R1** (an earlier January 2025 model) as a distillation teacher, proving that leaderboard accuracy alone does not predict how well a model's traces teach a student.
3. **Answer filtering did not help**: discarding teacher reasoning traces that arrived at the wrong final answer failed to improve student performance in their experimental setup.
4. **Curated quality beats raw breadth**: smaller, tightly curated high-quality sources (such as **OpenMath-2-Math**) outperformed larger, noisier, more diverse prompt mixtures.

COMMON MISCONCEPTION

You might think that when distilling reasoning traces, you should always pick the highest-ranked model on leaderboards as your teacher, sample one response per prompt to maximize prompt diversity, and strictly delete every trace whose final answer is wrong. OpenThoughts' ablations refuted all three assumptions in their setup: QwQ-32B beat earlier model DeepSeek-R1 as a teacher, generating 16 responses per prompt helped significantly, and answer-based filtering failed to improve downstream accuracy. Never assume a post-training recipe transfers without ablating teacher choice, sampling multiplicity, and filtering.

Software Engineering (SWE) Data: From Semi-Synthetic Tasks to Execution-Free Scaling

Software engineering (SWE) tasks—fixing bugs or implementing features across multi-file repositories—expose the sharpest infrastructural bottlenecks in post-training data creation. Unlike math problems or isolated coding contests, real repositories depend on complex, fragile operating-system packages and library versions; building and maintaining thousands of working **Docker** containers is a major engineering bottleneck. Four recent systems illustrate the progression of strategies for scaling SWE post-training data.

1. Semi-Synthetic Tasks: SWE-smith

SWE-smith solves the container bottleneck with a **semi-synthetic** recipe: keep a real, working software repository and Docker environment, and synthesize the bugs inside it.

DEFINITION

A **semi-synthetic task** pairs a real, unmodified environment (such as an existing GitHub repository and its test suite) with an artificially generated problem or defect (such as an LM-injected bug) whose solution remains automatically verifiable.

The SWE-smith pipeline flows across four boxes from left to right:

1. **Real GitHub Repo** (blue): start with a working repository container where existing **tests pass (✓)**.
2. **LM Injects Bug** (yellow): prompt a language model to edit functions or program logic to synthesize a realistic defect.
3. **Run Test Suite** (red): execute the repository's unit tests and **keep only edits that break existing tests (×)**.
4. **Verifiable Task** (green): emit the buggy repository state together with the original working code as the **gold fix** and the failing tests as the verifier.

Because setting up a single container enables generating hundreds of distinct bugs inside it, SWE-smith achieves high leverage per container: just **128 GitHub repositories** yield **50K verifiable SWE tasks**.

2. Scaling Real Interactive Environments: SWE-rebench

Synthetic bugs may differ subtly from the real pull requests (PRs) human developers write. **SWE-rebench** takes the **real-task** route by automating environment setup and screening across historical GitHub PRs in a three-stage horizontal pipeline:

1. **450K GitHub PRs** (gray): collect candidate pull requests from GitHub and GitHub Archive.
2. **Automated Setup & Screening** (yellow): use an LLM agent (**Qwen 2.5-72B-Instruct**, an earlier baseline) to automatically install repository dependencies and judge PR quality.
3. **21K Python SWE Tasks** (green): output verified interactive tasks spanning **3.4K GitHub repositories**.

Comparing SWE-rebench with SWE-smith highlights the cost of real tasks: screening 450K PRs across 3.4K fragile dependency environments yields 21K tasks—requiring far more container infrastructure per task than semi-synthetic generation.

3. Decoupling Execution: SWE-Zero and SWE-Hero

Must a teacher model execute live unit tests inside a Docker container during every single rollout in order to generate useful training trajectories? **SWE-Zero** answers **no**, resting on a key insight: strong coding models possess an internal **"world model"** of code semantics capable of diagnosing bugs and writing patches without live test execution.

Picture two stacked boxes comparing the hybrid strategy:

- **Top box (blue — SWE-Zero, No Repo Execution)**: takes **150K GitHub PRs** and generates **300K agent trajectories** using the **OpenHands** scaffold and **Qwen3-Coder-480B** as teacher, without setting up live execution containers during rollout generation. Critically, **future git commits are stripped** from the repository before the agent runs, and **post-hoc filtering** attempts to execute patches afterward where possible.
- **Bottom box (yellow — SWE-Hero, With Execution Feedback)**: generates **13K agent trajectories** on targeted tasks where live test execution feedback is essential.

> **DEFINITION**
> **Git hacking** is a form of data leakage in repository-level coding agents in which an agent inspects git commit history inside the workspace and peeks at future commits containing the ground-truth fix instead of solving the issue.

Data hygiene is decisive here: forgetting to strip future git commits lets teacher agents cheat via git hacking, polluting the trajectory corpus with useless shortcuts. Pairing high-volume execution-free trajectories (**SWE-Zero**) with targeted execution-verified trajectories (**SWE-Hero**) yields a practical hybrid strategy.

4. Scaling Trajectories: SWE-ZERO-12M-trajectories

Once rollout generation is decoupled from Docker execution setup, trajectory volume can be scaled by orders of magnitude. **SWE-ZERO-12M-trajectories** generates **12 million agent trajectories** across the **SWE-rebench-v2** task pool—which consists of **32K executable** tasks plus **120K non-executable** tasks—

using a compact, fast generator, **mini-coder-1.7b** (which achieves **50.4 pass@100**), running inside the **mini-swe-agent** scaffold.

This introduces a sharp engineering tradeoff: high-volume sampling from a small, fast generator unlocks millions of training trajectories even from non-executable repositories that container-based pipelines must discard—**provided post-hoc filtering reliably catches low-quality traces**.

> **WORKED EXAMPLE**
>
> **Container Leverage and Trajectory Sampling Economics Across SWE Pipelines**
>
> Compare the per-repository leverage and rollout statistics of SWE-smith, SWE-rebench, and SWE-ZERO-12M:
>
> 1. **Container leverage (tasks per Dockerized repository)**:
>   - **SWE-smith**: $50{,}000 \text{ tasks} / 128 \text{ repos} \approx 390.6 \text{ verifiable tasks per repo}$.
>   - **SWE-rebench**: $21{,}000 \text{ tasks} / 3{,}400 \text{ repos} \approx 6.2 \text{ tasks per repo}$, with an end-to-end PR yield of $21{,}000 / 450{,}000 \approx 4.7\%$. Semi-synthetic bug injection achieves over $63\times$ higher task yield per configured repository container than mining real PRs.
> 2. **SWE-ZERO-12M sampling density and filtering burden**:
>   - Total tasks in SWE-rebench-v2: $32{,}000 + 120{,}000 = 152{,}000$ tasks.
>   - Average trajectories generated per task:
>
> $$\frac{12{,}000{,}000}{152{,}000} \approx 78.9 \text{ trajectories per task.}$$
>
>   - Estimate the per-rollout solve rate q of `mini-coder-1.7b` from $\text{pass@100} = 0.504$. Modeling independent attempts gives $1 - (1-q)^{100} = 0.504$, so:
>
> $$(1-q)^{100} = 0.496 \implies q = 1 - 0.496^{1/100} = 1 - e^{\ln(0.496)/100} \approx 1 - e^{-0.00701} \approx 0$$
>
> **Interpretation:** Roughly 78.9 cheap rollouts per task turn even non-executable repos into a 12M-trajectory pool, but with a small generator solving on the order of 1% of individual attempts, the overwhelming majority of raw rollouts are unsuccessful—demonstrating quantitatively why post-hoc filtering makes or breaks execution-free scaling.

Post-Training Data Taxonomy and End-to-End Synthesis

Pulling these case studies together yields a clean taxonomy of post-training prompt and task paradigms:

Prompt / Task Paradigm	Examples	Strengths	Main Bottlenecks
Fully synthetic	Synthetic slices of OpenThoughts	Unlimited scale, full topic control	Diversity collapse, teacher bias
Semi-synthetic	SWE-smith (128 repos → 50K tasks)	Real codebases + verifiable tests	Synthetic bugs may differ from real PRs
Real tasks	SWE-rebench (21K), SWE-Zero (300K), SWE-ZERO-12M	Authentic developer distribution	Container setup, git hacking, noisy PRs

Across every paradigm, two overarching engineering principles hold: choose teachers empirically by student distillation gains rather than solely by leaderboard rank, and treat data hygiene (leakage, git hacking, and filtering) as first-class engineering because subtle bugs invalidate entire corpora.

Stepping back across the entire lecture—from **Transformation** (lossy HTML/PDF linearization via `trafilatura` / `resiliparse` and FinePDFs Docling/RolmOCR), through **Filtering** (target extrapolation with KenLM, fastText, and LLM-distilled classifiers tuned to token budget), **Deduplication** (linear-time hashing and MinHash + LSH banding around $t = (1/b)^{1/r}$), and **Data Mixing** (epoch caps via UniMax and scale-transferable RegMix via simulated epoching), to **Post-Training** (environments, real/semi-synthetic/synthetic tasks, and multi-teacher distillation)—one final principle unifies all five stages: **data work is deeply domain-specific, and there is no substitute for inspecting real examples at every stage of the pipeline.**

> **KEY IDEA**
> Post-training data combines environments, tasks (fully synthetic, semi-synthetic like SWE-smith, or real like SWE-rebench and SWE-Zero), and empirically chosen teachers—scaling in late-2026 frontier models to multi-teacher distillation—while execution-free rollout scaling (SWE-Zero / SWE-ZERO-12M) bypasses Docker bottlenecks at the cost of strict data hygiene against git hacking and heavy reliance on post-hoc filtering.

✎ Check Your Understanding

1. **Yes/No with justification:** Based on OpenThoughts' ablations, if Model A scores higher than Model B on reasoning benchmarks, can you safely assume Model A will produce better distillation traces for training a student model?
   Answer: No. OpenThoughts found that stronger benchmark performance does not guarantee a better teacher—QwQ-32B outperformed earlier model DeepSeek-R1 as a distillation teacher—so teacher choice must be ablated empirically.
2. **What would happen if…** you built an execution-free trajectory pipeline like SWE-Zero over historical GitHub PRs but forgot to strip future git commits from the repositories?
   Answer: The coding agent could commit "git hacking"—reading future commits in git history to peek directly at the ground-truth fix instead of reasoning through the bug—corrupting the training trajectories with data leakage.