Mixtera: A Data Plane for Foundation Model Training

Maximilian Böther
mboether@ethz.ch
ETH Zurich
Switzerland

Xiaozhe Yao
xiaozhe.yao@ethz.ch
ETH Zurich
Switzerland

Tolga Kerimoglu
tkerimoglu@student.ethz.ch
ETH Zurich
Switzerland

Dan Graur
dan.graur@ethz.ch
ETH Zurich
Switzerland

Viktor Gsteiger
vgsteiger@student.ethz.ch
ETH Zurich
Switzerland

Ana Klimovic
aklimovic@ethz.ch
ETH Zurich
Switzerland

Abstract

State-of-the-art large language and vision models are trained over trillions of tokens that are aggregated from a large variety of sources. As training data collections grow, manually managing the samples becomes time-consuming, tedious, and prone to errors. Yet recent research shows that the data mixture and the order in which samples are visited during training can significantly influence model accuracy. We build and present Mixtera, a data plane for foundation model training that enables users to declaratively express which data samples should be used in which proportion and in which order during training. Mixtera is a centralized, read-only layer that is deployed on top of existing training data collections and can be declaratively queried. It operates independently of the filesystem structure and supports mixtures across arbitrary properties (e.g., language, source dataset) as well as dynamic adjustment of the mixture based on model feedback. We experimentally evaluate Mixtera and show that our implementation does not bottleneck training and scales to 256 GH200 superchips. We demonstrate how Mixtera supports recent advancements in mixing strategies by implementing the Adaptive Data Optimization (ADO) algorithm in the system and evaluating its performance impact. We also show how Mixtera enables exploring the role of mixtures for vision-language models, which is a growing area of research.

ACM Reference Format:
Maximilian Böther, Xiaozhe Yao, Tolga Kerimoglu, Dan Graur, Viktor Gsteiger, and Ana Klimovic. 2026. Mixtera: A Data Plane for Foundation Model Training. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 54 (February 2026), 16 pages. https://doi.org/10.1145/3786668

1 Introduction

Large language and vision models (LLMs/VLMs, often called foundation models) have become omnipresent in our daily lives. They show enormous capabilities in a diverse set of tasks [9, 11, 39, 47, 53], such as assistance with writing and coding, video understanding, and even agentic interaction with the world. The training of such language and vision models presents new challenges for managing training data due to the ever-growing sizes of models and datasets. To achieve high accuracy, state-of-the-art models train over trillions of tokens. For example, Meta's Llama 3.3 70B model is trained on a corpus of 15 trillion tokens [39, 40]. These tokens typically come from aggregated data collections such as RedPajama [70], Dolma [62], or FineWeb [49], which include data from various sources, such as Wikipedia or Common Crawl dumps.

The composition of training data is critical to model quality [12]. Hence, selecting the right proportions of data with particular characteristics (e.g., language, topic, source) has become an active area of research to improve model performance without increasing training compute budget [14, 71, 75]. For example, Hugging Face's SmolLM2 model is trained with four stages of data mixtures that combine web text with specialized math, code, and instruction-following data in varying proportions [4]. Algorithms such as Adaptive Data Optimization (ADO) [28], Aioli [13], PiKE [33], and Skill-It [14] even propose adjusting the data mixture *dynamically* based on the model behavior (e.g., loss per domain) during training. Figure 1 shows that ADO increases the accuracy of 1.6B and 3.6B Llama-models (c.f. Table 2) compared to using a static mixture on the downstream HellaSwag benchmark [78].

However, the process of composing training data mixtures today is manual, ad hoc, and error-prone (Figure 2a). Training data is typically stored on distributed filesystems in GPU clusters or data lakes in the cloud. ML engineers and researchers write ad hoc scripts to process the training data, filter relevant subsets with the properties of interest, often pre-tokenize it, and then mix it for their use case. For example, a model developer may want to train on 50 % data from Wikipedia and 50 % from movie subtitles. This can quickly become more complex as training data may need to be mixed based on multiple characteristics. For example, in addition to source data

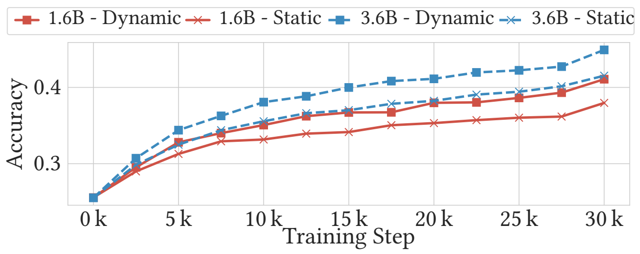
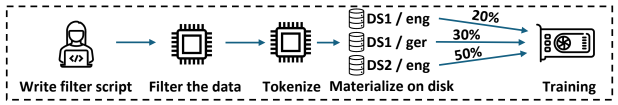


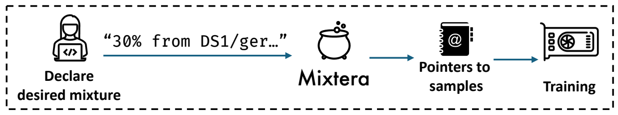
Figure 1: Dynamically adjusting the mixture using the ADO algorithm improves pre-training performance on HellaSwag over the default static mixture across model scales.

Authors' Contact Information: Maximilian Böther, mboether@ethz.ch, ETH Zurich, Switzerland; Xiaozhe Yao, xiaozhe.yao@ethz.ch, ETH Zurich, Switzerland; Tolga Kerimoglu, tkerimoglu@student.ethz.ch, ETH Zurich, Switzerland; Dan Graur, dan.graur@ethz.ch, ETH Zurich, Switzerland; Viktor Gsteiger, vgsteiger@student.ethz.ch, ETH Zurich, Switzerland; Ana Klimovic, aklimovic@ethz.ch, ETH Zurich, Switzerland.

ACM 2836-6573/2026/2-ART54
https://doi.org/10.1145/3786668



(a) Without Mixtera: More developer, CPU, and disk resources needed.



(b) With Mixtera: Declarative mixture specification and low overhead.

Figure 2: Data preparation workflows.

proportions (Wikipedia vs. movie subtitles), the developer may also want the training data to be 80 % in English and 20 % in German. When using mixing algorithms such as ADO, the data mixture may also need to be adjusted on the fly during training.

Model developers currently lack an open-source solution to efficiently manage and declaratively query vast amounts of training data based on the characteristics of individual data elements. This reflects our experience working with ML researchers as part of a large-scale initiative developing open-source LLMs and our conversations with industry teams. Implementing data filtering and subsequent mixing requires today's model developers to manually keep track of metadata. At least in the open-source world, developers typically implement this as part of the directory structure of the filesystem, e.g., developers create one subdirectory per source dataset and then sample from each directory (Figure 2a). This approach is limited because each data sample has multiple properties that can be used to determine whether it should be used for training. Filesystems fundamentally do not offer the right interface for managing training data and mixing, as they do not provide declarative query interfaces or a native way to track which model was trained on what data. Running an offline data processing job with frameworks like Apache Beam or Spark to fully materialize the mixed training set for each training run makes it difficult to quickly iterate on different mixtures during the exploration phase and leads to data duplication, increasing storage costs. While databases offer a declarative query interface, using a fully-fledged DBMS to track data properties would burden ML engineers with database administration, schema design, and database performance tuning. Having a simple read-only SQLite table at each training client might seem like a straightforward solution, but it cannot dynamically adjust data mixtures and it does not interface with training frameworks like Torchtitan [34]. We need a flexible data plane that lets users easily find and combine data based on any characteristic they choose, without being restricted by how files are organized. This system should also allow users to change the data mixture on the fly and work seamlessly with current training frameworks.

We present Mixtera, a data plane that can be deployed on top of existing LLM/VLM training data collections stored in distributed filesystems or cloud data lakes. Mixtera follows a client-server model. The user indexes all sample metadata at the server once. During a training job, the server statically filters out the relevant samples using SPJ-style predicates, and then continuously distributes *chunks* to the clients. Chunks are fixed-size collections of *pointers* to samples (Figure 2b) in files that adhere to the current mixture. The clients fetch the relevant samples and return the relevant data to the training loop. By storing metadata, and deferring reads to the clients, Mixtera scales to real-world datasets. Chunks also enable dynamic data mixing and avoid re-materializing mixed data. This approach also improves the iteration speed for model developers. While existing data loaders, as outlined in Table 1, rely on sampling from filesystem directories, Mixtera supports arbitrary properties independent of the filesystem, and makes it easy for model engineers to experiment with different filter criteria, fixed learning curricula [4, 72], and fully dynamic mixing algorithms that learn the mixture during training [13, 14, 28]. We design and implement Mixtera tailored to the needs of foundation model training, and contribute the following:

(1) Mixtera indexes all samples and their properties, enabling users to declaratively specify mixtures across properties independent of the filesystem structure and training frameworks.
(2) Mixtera enables dynamically changing the data mixture and the properties used for mixing. It achieves this by generating and streaming chunks, i.e., fixed-size lists of pointers to samples following the current mixture.
(3) Mixtera's data fetching scales to meet the ingestion throughput demands of large-scale training jobs. We run benchmarks spanning 256 GH200 superchips and show that Mixtera does not limit training throughput. As an example for how it enables model accuracy improvements, we demonstrate how to implement the ADO dynamic data mixing algorithm in Mixtera and its positive impact on model accuracy.

2 Background

Foundation models are large-scale deep learning models suitable for a variety of tasks [9, 17]. We focus on text-generation models, i.e., autoregressive large language models (LLMs) and multimodal vision-language models (VLMs). As of 2025, most such models are based on the Transformer architecture [68]. They are trained on vast corpora of training data in a self-supervised manner to maximize the likelihood of predicting the tokens of a training sequence.

Training phases. Training is structured into *pre-training* and *post-training* phases. In pre-training, we train a randomly initialized model on a general-purpose data corpus (Section 2.1) to derive a *base model*. In post-training, common steps include *supervised finetuning* (SFT) and *alignment*. In this paper, we perform only pre-training experiments due to space constraints, but Mixtera can also be used for post-training.

Distributed training. Training foundation models requires distributing computation across multiple GPUs. Training frameworks typically employ 3D parallelism [6, 23], consisting of pipeline parallelism (PP), i.e., partitioning the model layers between devices [24, 45, 46], tensor parallelism (TP), i.e., splitting individual tensor operations within layers across devices [57, 59], and data parallelism (DP), i.e., replication of the model across device groups. PP and TP together are referred to as *model parallelism*. Nodes within the same DP group process identical inputs, while nodes across DP groups receive different data. As an extension to DP, fully-sharded data parallelism (FSDP) shards model parameters, gradients, and optimizer states across data-parallel workers [82].

2.1 Training Data and Data Mixing

Pre-training data stems from data collections that include samples from various sources (e.g., Wikipedia, Common Crawl dumps, or arXiv papers). Public examples of such collections include RedPajama [70], Dolma [62], and FineWeb [49]. Besides aggregating data from different sources, data engineers typically clean the data, which usually involves deduplicating, filtering (e.g., removing personal identifiable information), and applying classifiers to the data samples (e.g., to obtain a toxicity score for each sample) [37, 49].

Data properties and mixtures. Each data sample has properties, such as its source (e.g., Wikipedia) or its language (e.g., English). ML engineers need to define a *data mixture*, which describes how the data is combined based on its characteristics, e.g., we can train on 50 % data from Common Crawl and 50 % from movie subtitles. The data can be combined based on multiple characteristics simultaneously. For instance, besides Common Crawl and movie subtitles, we might also use 80 % French and 20 % Italian data.

Mixing algorithms. Selecting the best mixture is critical for model performance [14, 58, 71, 75]. We differentiate *static mixtures*, i.e., mixtures that remain constant over the entire training job, and *dynamic mixtures*, i.e., mixtures that change during the training job. Recent research proposes several algorithms for finding the best static mixture or how to adjust the mixture dynamically during training. Algorithms such as DoReMi [71] or the data mixing laws [75] find a static mixture via small proxy models.

Curriculum learning is an example of a pre-defined dynamic mixture. Xu et al. [72] order samples from easy to hard to improve alignment. The SmolLM2 model was trained on 4 stages of mixtures [4]. Multilingual models are often first trained on English data, followed by samples from other languages [54, 73].

Beyond such pre-defined schedules, for text-only models, there is also work on adapting the data mixture to the model training dynamics, e.g., by increasing the weight of data domains that have high loss. Albalak et al. [3] model mixture components as arms of a multi-armed bandit. Skill-it orders "skills" based on model feedback [14]. Aioli builds upon Skill-it and provides a unified framework for estimating the best mixture during training [13]. PiKE relies on gradient interactions [33]. In this paper, we use Adaptive Data Optimization (ADO) [28] as an example of a dynamic mixing algorithm.

2.1.1 Adaptive Data Optimization. Adaptive Data Optimization (ADO) is a dynamic mixing algorithm that adjusts the data mixture during training based on the model's learning progress on each domain [28]. The key idea is to prioritize domains where the model shows rapid improvement while considering how much each domain benefits from its own samples. ADO uses neural scaling laws to model how the loss L_k of each domain k decreases with the number of training samples n. To this end, it fits a power law $\hat{L}_k(n) = \varepsilon_k + \beta_k n^{-\alpha_k}$ *for each domain.* Here, ε_k represents the irreducible loss of the domain, β_k is a scaling factor, and α_k determines how quickly the loss decreases. The parameters are re-fitted during training. The algorithm combines two components to determine the mixture weights. First, it estimates the learning speed for each domain using the derivative of the scaling law. Second, ADO maintains a credit assignment score $\lambda_k(t)$ that indicates how much each domain contributes to its own progress, based on its recent sampling frequency. These components are combined with a prior (initial) distribution μ_k to compute an intermediate preference distribution $\rho_k(t)$. To ensure stability, the final distribution $\pi_k(t)$ is then computed as a weighted average between $\rho_k(t)$ and $\pi_k(t)$'s temporal average. Additionally, ADO enforces a minimum sampling probability for each domain.

3 Current Challenges

We identify three challenges in the status quo of training data management with current open-source infrastructure.

Challenge 1: Today's training data storage systems lack expressive, declarative APIs for data mixing. Training data is typically stored and managed as files on distributed filesystems or objects on cloud storage. As these systems are not natively built for foundation model training data [69], they lack operators for data selection, mixing, and ingestion into a training framework. This also complicates lineage tracking, as there is no native way of tracking which model was trained on which data when the data is accessed via general-purpose filesystem calls.

Challenge 2: Preparing and materializing data mixtures with offline preprocessing limits flexibility and increases storage costs. The current approach to preparing training data involves numerous manual offline steps with general-purpose data processing and scripting frameworks (Figure 2a). For the offline cleaning step (Section 2.1), ML engineers typically leverage data processing frameworks like Spark [77], Beam [1], Data-Juicer [12], or datatrove [50]. Subsequent data mixing can happen offline or online. Current online data loaders sample data from directories that reflect the mixing property, e.g., one directory per source dataset (Table 1), and therefore do not support switching the property we mix on nor specifying hierarchical mixtures across arbitrary properties. Due to the limited functionality of current online solutions, engineers often write ad hoc offline mixing scripts that create a new mixed copy of the cleaned dataset for each training run, which can

Table 1: Feature comparison of Mixtera and other open-source data loaders.

	Mixtera	HF Datasets	WebDatasets	Mosaic Streaming
File formats	jsonl(.zst), parquet, webdataset	jsonl(.zst), parquet, webdataset, csv	webdataset	Mosaic Data Shard, jsonl, csv
Static filtering	declarative	using map UDFs	using map UDFs	using map UDFs
Static mixtures	on all properties	on filesystem dirs.	on filesystem dirs.	on filesystem dirs.
Dynamic mixtures	on all properties	no	no	no
Native 3D parallelism	yes	yes, manual rank handling	data parallel only	yes, for specific rank order
Checkpointing	yes	using TorchData	by replaying, not natively	yes

drastically increase storage usage and cost. As some frameworks like Megatron [46, 59] even require pre-tokenized data, this leads to duplication of the even bigger tokenized data files. Offline mixing also drastically increases the iteration time and makes it difficult to get a quick sense of how a mixture will impact model training when exploring different mixing policies.

Challenge 3: Lack of support for dynamic mixtures during training. How to train the best model on a given dataset is an active research area, with dynamic mixing emerging as a prominent new technique. Offline preparation of the mixture or other approaches such as online mixing based on fixed directory weights or using a vanilla DBMS without additional infrastructure does not support dynamic mixture at all. Additionally, even for researchers who are familiar with the latest mixing techniques, implementing modern mixing algorithms in a training pipeline is a painful, tedious, and error-prone task, as the codebases for mixing algorithms are often tailored directly to the training framework, as well as the data collection and properties used in the respective papers. This hinders the adoption of dynamic mixing algorithms and makes researching, reproducing, and comparing difficult.

4 Mixtera's Design

We address these challenges for training data management by building Mixtera, a foundation model training data plane. We derive the following design goals for such a system.

Goal G1: The system should implement a *centralized* data layer that users can conveniently and declaratively query to mix data across arbitrary properties, independent of the filesystem structure.

Goal G2: The system needs to be *lightweight*, i.e., easily integrate into existing training deployments without requiring setting up many components.

Goal G3: The system needs to ensure *high-throughput*, *determinism*, and *reproducibility*, while being user-friendly and flexible.

Goal G4: The system must support *adjusting the mixture dynamically* during training.

4.1 Data Model

Mixtera operates on training datasets of the following structure:

Sample. A sample s is the atomic unit of training data, *as supplied by the user*. In LLM training, the *sample unit* is typically a text string or a pre-tokenized sequence; VLM training uses multimodal samples (e.g., text-image pairs). Sample granularity varies in practice, ranging from short texts to full documents.

Property. A property p is a named attribute of a sample, with a domain $\mathcal{D}_p$ of possible values. Properties characterize samples and can be **single-valued** (each sample has exactly one value, e.g., $p_{\text{lang}} \in \{\text{English, French, German}\}$), or **multi-valued** (each sample may have multiple values, e.g., $p_{\text{topic}} \subseteq \{\text{science, politics}\}$).

Data Collection. A data collection C is a tuple $(\mathcal{S}, \mathbb{P}, \phi)$ where $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ is a finite set of samples, $\mathbb{P} = \{p_1, p_2, \dots, p_m\}$ is a set of properties, and $\phi : \mathcal{S} \times \mathbb{P} \to \mathcal{P}(\mathcal{D}_p)$ maps each (sample, property) pair to a subset of values from that property's domain. We also refer to it as the sample metadata.

Mixtera defines mixtures fully independent of the filesystem–it is fully based on (logical) properties.

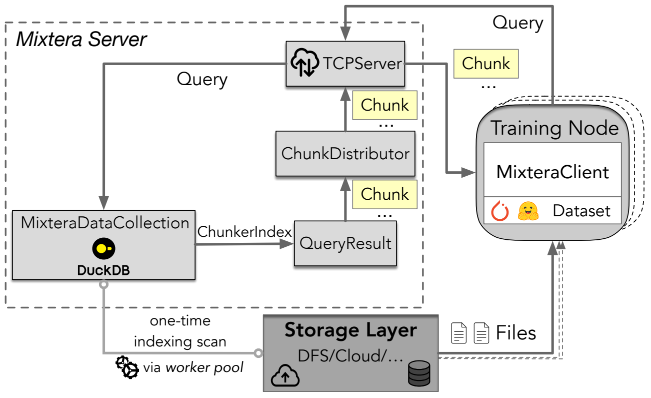


Figure 3: Mixtera system architecture.

Mixture Key. A mixture key k is a partial mapping from a subset of properties $\mathbb{P}' \subseteq \mathbb{P}$ to sets of values, i.e., $k : \mathbb{P}' \to \bigcup_{p \in \mathbb{P}'} \mathcal{P}(\mathcal{D}_p)$, with $\forall p \in \mathbb{P}' : k(p) \subseteq \mathcal{D}_p$. For mixture keys k_1 and k_2, we say k_1 **matches** k_2, if f.a. $p \in \text{dom}(k_1), p \in \text{dom}(k_2) \wedge k_1(p) \cap k_2(p) \neq \emptyset$. This relation is non-symmetric.

Component Key. A sample $s \in \mathcal{S}$ induces a component key $\hat{k}_s$, s.t. $\forall p \in \mathbb{P} : \hat{k}_s(p) = \phi(s, p)$.

Data Mixture. A data mixture M is a mapping from mixture keys to target proportions: $M : \mathcal{K} \to [0, 1]$ with $\sum_{k \in \mathcal{K}} M(k) = 1$.

4.2 System Overview

Mixtera is a read-only layer that can be deployed on top of existing sets of samples $\mathcal{S}$, which are typically stored in a distributed filesystem or cloud object store. Figure 3 shows the client-server architecture of the system. The server runs on one node, and each training node runs client instances. Mixtera manages a centralized database which stores the mapping ϕ, i.e., the sample metadata (G1). This database needs to be populated by scanning all samples once before it can be used for training. Since the metadata typically consists of integers and floats, it is much smaller than the actual samples. Mixtera defers the reading of actual sample payloads to clients. This enables to scale to large-scale training datasets.

Mixtera assigns every sample a unique ID. It allows model developers to declaratively query the relevant samples for a training job. To remain lightweight (G2), Mixtera does not reorganize or modify the data files on disk. It provides a standard iterator for model training that can be used in conjunction with loaders like `torch.DataLoader`. Mixtera is agnostic to the training framework, supports training interruptions using checkpoints, and ensures determinism (G3) through careful shuffling, i.e., for identical queries, Mixtera always provides data in an identical order, which is important for reproducibility and debugging issues like loss spikes [18, 29, 51, 66, 83]. It supports adjusting the mixture during training (G4) by transferring chunks (lists of pointers to samples) whose mixture can change over time.

Data types. Mixtera supports diverse sample types (e.g., pre-tokenized text in binary files, strings in `jsonl`/ `parquet`, text–image pairs in `webdatasets`). This covers many relevant use cases. Even modern foundation models for tabular data treat tables as strings [8]. The sample granularity is inherent to the ingested dataset. As Mixtera is a read-only layer, it adopts whatever granularity the data

```python
client = MixteraClient("127.0.0.1", 8080)
job_id = "test_job"
query = Query.for_job(job_id).select(("license","==","CC"))
mixture = StaticMixture(
    { MixtureKey({"language": ["JavaScript"]}): 0.7,
     MixtureKey({"language": ["HTML"]}): 0.3 },
    chunk_size=1024)
qea = QueryExecutionArgs(mixture=mixture, num_workers=4,
                         dp_groups=1, nodes_per_group=1)
rsa = ResultStreamingArgs(node_id=0, dp_group_id=0, job_id=job_id)
ds = MixteraTorchDataset(client, query, qea, rsa)
dl = torch.utils.data.DataLoader(ds, batch_size=1024, num_workers=4)

for batch in dl:
    print(batch)
```

Figure 4: An example query using Mixtera.

already uses. While some features such as tokenization or image decoding are modality-specific, the system is fundamentally agnostic to the sample unit.

Chunks. Chunks are Mixtera's core abstraction for scheduling and mixture enforcement. *Abstractly*, a chunk C_h is a collection of samples $\{s_1, \ldots, s_c\}$ of fixed size $c = |C_h|$, and the set conforms to the current mixture M. That is, for each mixture key $k \in \mathcal{K}$, the proportion of samples in C_h matching k is approximately $M(k)$. *In implementation*, Mixtera does not materialize or transfer the sample payloads $\mathcal{S}$. Instead, chunks contain only *pointers* to samples: metadata specifying which samples in which files to load (e.g., "samples 100-150 in `wikipedia.jsonl.zst`").

Deferred reading. Storing only metadata and distributing pointers to samples (as opposed to ingesting the actual sample payloads into the system) has several advantages. First, users can store data in their locations of choice (e.g., an object store, or a distributed filesystem). Chunks are independent of the filesystem structure (G1). Second, it allows Mixtera to support dynamic mixtures (G4), as the data composition of chunks can change over time. Third, the pointer-model avoids creating a data fetching bottleneck at the Mixtera server. The server only creates chunks and each client fetches the data they need. Fourth, we avoid a lock-in effect and allow for easy adoption on existing data collections (G2). Last, we natively support other modalities (whereas image or video payloads would not be straightforward to ingest at scale into a database).

Query interface. In Figure 4, we show an example query that statically selects only Creative Commons data, and then mixes HTML and JavaScript data in a 70:30 ratio during training. Mixtera takes care of executing the query and obtaining the samples without needing to worry about correctness, even in distributed training. The user only needs to provide the ID of the node and its data parallel group, which is obtained from the training framework.

Mixtera allows expressing static filter operations on properties via SPJ-style predicates, and static as well as dynamic mixtures across all properties (G4). The static filter defines the ground set of all potential data (e.g., only CC), while the mixture describes the proportions of how the data from this set is mixed during training (JavaScript:HTML ratio). The `MixtureKey` class operationalizes its formal definition (Section 4.1) in a user-friendly Python API.

Executing a query. Before submitting a query, users catalog all samples at the server. This involves defining the data schema as a Python class, which allows Mixtera workers to scan all samples and populate the database. Users then can submit queries. A query is executed at the server in two phases. First, Mixtera applies static filters from the query (e.g., English-only) to obtain all samples eligible for training (`QueryResult`). Second, during training, the server distributes *chunks* of that query result to the client(s), which specify which samples to train on. The server ensures that the chunks are distributed correctly, i.e., tensor- and pipeline parallel stages receive the same input data. The server generates chunks according to the current mixture, i.e., it iteratively takes samples from the query result such the chunk abides by the current mixture. As an iterable data loader, Mixtera faces the challenges of determinism and checkpointing. We address this by shuffling based on the query and support to load/store the query state.

High-throughput data fetching. The challenge with not reorganizing the user's data files is that Mixtera needs to handle suboptimal data layout. Files may have arbitrary distributions of data properties (e.g., even if a file only contains Wikipedia data, data in different languages might be distributed randomly across files). Formats like `jsonl` were not built with random access in mind, yet chunks force clients to load individual samples from files. To avoid data stalls (G3), Mixtera reads subsequent samples in files if the samples follow the same properties (e.g., same language). This is achieved through intervals $(f, [i, j], \hat{k})$ identifying consecutive samples in file f that share the same component key $\hat{k}$. A chunk is thus represented as a collection of intervals whose total size equals c and whose union satisfies the mixture. We use nested CTEs in DuckDB to find these intervals as fast as possible (Section 5.3). Users are not required to use sequential file formats.

Alternative design considerations. A metadata SQLite database at each client might seem like a simple alternative. While DBs can federate and query external sources with SPJ-style predicates, we still need an interface between the storage, query engine, and training framework, which can turn query results into input tensors (G1, G2). Furthermore, using a database at each client would not enable dynamic mixing algorithms (G4). Mixtera serves as the *interface* that connects a metadata database to the training framework, with a full feature set for dynamic model training. It also alleviates ML engineers from manually maintaining the database. We do not find centralized metadata management to be a bottleneck even for large-scale training jobs, because the work at the service is minimal (occasional chunk creation and distribution). Mixtera maintains high training throughput for large training jobs (Section 6.3). Industry frameworks like Google's Pathways [5] also rely on a single Python process to coordinate large-scale training. A current limitation of Mixtera's design is that it does not enable incorporating new data into an existing query. However, this (i) affects nearly all dataloaders as they all build some internal state, and (ii) and can in all cases be dealt with by executing a new query, e.g., in continual pretraining from a model checkpoint.

Open source ecosystem. Mixtera comes as a Python package that provides the entry point for the server and abstractions for the client. The codebase, consisting of approximately 11 k lines of Python and C++ (excluding tests), is open-source[1]. It is rigorously tested with a full set of unit and integration tests. We are continuing to add features and welcome contributions.

[1]Available at https://github.com/eth-easl/mixtera.

5 Implementation

We explain how Mixtera ingests sample metadata (Section 5.1), how it executes queries and creates chunks (Section 5.2), how those chunks are parsed at the client (Section 5.3), and describe Mixtera's integration into training frameworks (Section 5.4).

5.1 Metadata Insertion

Mixtera manages the metadata ϕ in an `MixteraDataCollection` (MDC) which uses DuckDB [52] as the underlying DBMS to efficiently store and query sample properties. The implementation itself is agnostic to the DBMS. While DuckDB's pluggable extension architecture (or similar approaches like PostgreSQL's Foreign Data Wrappers or SQLite's Virtual Tables) would allow to query the sample files directly, to defer reading (Section 4.2) to the training nodes, the MDC serves as a metadata index.

Initial ingestion. To populate the MDC, users need to define a `MetadataParser`. A `MetadataParser` operationalizes the property set $\mathbb{P}$ and extraction of $\phi(s, p)$ for each sample s. It defines the schema specifying $\mathbb{P}$ and domains $\mathcal{D}_p$ for each property. In Python, the schema is a list of properties which have a type (e.g., string or enum), a nullable field, and a multiple field, describing whether a single sample can take multiple values for this properties (e.g., several languages). Mixtera adjusts the underlying database table by mapping the Python schema to a proper database schema. The system comes with a set of pre-defined parsers for common datasets and enables users to define custom parsers.

File scanning. Metadata ingestion is a one-time preprocessing step. The MDC first accumulates all data files and prepares the database schema. Mixtera then parallelizes metadata extraction using a worker pool. Workers process files in batches, where each worker sequentially reads samples from its assigned files and applies the `MetadataParser` to extract property values (e.g., parsing JSON fields). Since DuckDB does not support concurrent insertions from multiple processes, Mixtera aggregates worker results in the main process. To optimize insertion throughput, we convert the collected metadata to columnar PyArrow in-memory tables before bulk insertion into DuckDB. Importantly, metadata extraction is *decoupled* from training. Users must define properties upfront, but can later add new properties by re-scanning the dataset with an updated parser without affecting existing metadata.

5.2 Server-Side Query Execution

After registering data, users can execute queries. When a client sends a query, the server executes it in two phases. The first phase is performed via the MDC and applies static filters to identify all relevant samples, and groups consecutive samples into intervals. The second phase constructs a data structure called `ChunkerIndex` that enables efficient, mixture-aware chunk generation.

5.2.1 SQL generation and interval detection. After receiving an object representation of the query (c.f. Figure 4), similar to ORM frameworks like sqlalchemy, the Mixtera server generates a base SQL query from this object. This query returns a table in which each row represents a sample that the user is interested in. Mixtera ensures that the generated SQL matches the MDC's table schema, e.g., whether a property can have multiple values or not.

A key challenge for Mixtera is efficient random access to samples within files. File formats like jsonl or parquet are optimized for sequential reading rather than random access. To address this, Mixtera implements an interval-based approach: the server wraps the base filtering query in an outer query that identifies continuous ranges of samples sharing identical properties within the same file. Consider the following example result of a base filtering query:

Sample ID	File ID	Language	License
1	1	JavaScript	MIT
2	1	JavaScript	MIT
3	1	JavaScript	MIT
4	1	Python	Apache
5	1	Python	Apache
1	2	Python	Apache

Instead of treating these as six individual samples, Mixtera identifies three intervals:

- Interval 1: Samples 1-3 (File 1, JavaScript, MIT)
- Interval 2: Samples 4-5 (File 1, Python, Apache)
- Interval 3: Sample 1 (File 2, Python, Apache)

Even though samples 4-5 (file 1) and 1 (file 2) share the same properties (Python, Apache), they are in different files and thus form separate intervals. The primary key is formed by the sample and file ID. Mixtera constructs a SQL query that processes the data in multiple stages:

(1) First, Mixtera establishes a Common Table Expression (CTE) named `base_data` that contains the filtered samples:

```
WITH base_data AS (
    -- Our generated base filtering query here, e.g.,
    SELECT * FROM samples WHERE license = 'MIT'),
```

(2) Next, Mixtera identifies breaks in the sample sequence using window functions. The `grouped_samples` CTE calculates the difference between consecutive sample IDs within groups sharing the same properties:

```
grouped_samples AS (
    SELECT *, sample_id - LAG(sample_id, 1, sample_id)
    OVER (PARTITION BY file_id, lang, license
            ORDER BY sample_id) AS diff
    FROM base_data),
```

Here, a `diff` value of 1 indicates consecutive samples, while any other value indicates a break in the sequence.

(3) The `intervals` CTE then groups the sequences into intervals:

```
intervals AS (
    SELECT file_id, lang, license,
        SUM(CASE WHEN diff != 1 THEN 1 ELSE 0 END)
            OVER (PARTITION BY file_id, lang, license
                ORDER BY sample_id) AS group_id,
        MIN(sample_id) as int_strt, MAX(sample_id)+1 as int_end
    FROM grouped_samples
    GROUP BY file_id, lang, license, diff, sample_id)
```

The `group_id` is incremented when there is a break in the sequence, creating unique identifiers for each interval.

(4) Finally, Mixtera aggregates the results to get the final intervals:

```
SELECT file_id, lang, license, group_id,
  MIN(int_strt) as interval_start, MAX(int_end) as interval_end
FROM intervals
GROUP BY file_id, lang, license, group_id
ORDER BY file_id, interval_start;
```

Using intervals of samples can only improve I/O if samples within files are clustered by properties and not randomly distributed. Since Mixtera is read-only by design, it does not re-shuffle data.

5.2.2 *Chunk generation.*

After obtaining the query result with all relevant intervals, the server next runs the chunk generation algorithm. This algorithm is based on the `ChunkerIndex` data structure, which organizes sample ranges by their properties. We re-visit the `MixtureKey` concept from an implementation perspective.

`MixtureKey` abstraction. A `MixtureKey` represents a set of properties and their values. The class implements the formal definition from Section 4.1: a partial mapping $k : \mathbb{P}' \to \bigcup_{p \in \mathbb{P}'} \mathcal{P}(\mathcal{D}_p)$ for some $\mathbb{P}' \subseteq \mathbb{P}$. The matching relation enables flexible querying: k_1 matches k_2 if $\forall p \in \text{dom}(k_1) : p \in \text{dom}(k_2) \wedge k_1(p) \cap k_2(p) \neq \emptyset$. This matching is crucial as the resulting interval table from DuckDB contains the full cross-product of all properties—a sample might have values for language, license, size, topic, and more—while a mixture specification may consider only a subset of these properties, as we discuss in the next paragraph. It also allows us to define mixtures on *multiple properties* with *multiple values*, instead of being limited to a single property (c.f. Section 3). To ensure deterministic behavior, we implement a total ordering over keys based on the number of properties, property names, and their values. We sometimes refer to a specific `MixtureKey` as a *domain*, e.g., the key for `lang:English` defines the domain of English samples.

The `ChunkerIndex`. Recall from Section 4.1 that each sample $s \in \mathcal{S}$ induces a component key $\hat{k}_s$ defined f.a. $p \in \mathbb{P}$ as $\hat{k}_s(p) = \phi(s, p)$. The `ChunkerIndex` organizes intervals by these component keys, i.e., ChunkerIndex : $\left\{\hat{k}_s \mid s \in \mathcal{S}'\right\} \to \text{DatasetID} \to \text{FileID} \to \text{List}\left[[i, j)\right]$, where $\mathcal{S}'$ is the filtered subset from the query, and intervals $[i, j)$ group consecutive samples sharing the same $\hat{k}$. While the index maintains the complete property information of samples, it enables efficient sample lookup: given a mixture key k, we identify all component keys $\hat{k}$ where k matches $\hat{k}$, and retrieve their associated intervals. Consider a simplified example with `MixtureKeys` as strings. A fragment of the `ChunkerIndex` might look like:

```
{ "language:JavaScript,HTML;license:MIT": {
    ds_1: {
      file_1: [(1,4), (10,15)],  # half (right) open ranges
      file_2: [(1,2)]
    }},
  "language:Python;license:Apache": { ds_1: { file_1: [(1,2)] }}}
```

In this example, a query for `language:JavaScript` would match the first key despite it having the additional license property and two assigned languages. This demonstrates how the `MixtureKey` matching allow to work with the full property/value cross-product in the index while supporting mixtures on subsets of properties.

Building the ChunkerIndex. The index is built in parallel in a C++ extension, processing the interval table from DuckDB provided in Apache Arrow format. Operating on the Arrow table in Python would be too slow due to Global Interpreter Lock (GIL) constraints. Using `multiprocessing` to circumvent the GIL would require expensive pickling of nested dictionaries. Our C++ implementation uses multithreading and only acquires the GIL at the end.

Each C++ worker thread maintains a local index for a subset of the data. For each row (interval), each worker constructs a C++-representation of the `MixtureKey`, inserts the interval into its local index under this key, maintaining sorted order within each file's interval list. After parallel processing, the local indices are merged, combining interval lists while preserving their sorted order. In the end, we convert the index to Python objects, which often is the most expensive operation of this process.

Algorithm 1: Chunk generation algorithm. Some early exits and details are omitted for readability.

```
1  Initialize remaining_counts from mixture;
2  chunk ← ∅;
3  progress ← true;
4  while ∃key : remaining_counts[key] > 0 and progress do
5      progress ← false;
6      foreach mixture_key in remaining_counts do
7          foreach component_key in chunker_index do
8              if mixture_key matches component_key then
9                  Take up to remaining_counts[mixture_key]
                     samples from
                     chunker_index[component_key];
10                 if got > 0 samples then
11                     Add samples to chunk;
12                     Update remaining_counts;
13                     progress ← true;
14         if remaining_counts[key] > 0 and is best-effort then
15             Redistribute remaining counts to other keys;
16 if ∀key : remaining_counts[key] = 0 then
17     return chunk;
```

Chunk generation. As defined in Section 4.2, a chunk C_h is abstractly a collection of c samples conforming to mixture $M : \mathcal{K} \to [0, 1]$. In implementation, C_h contains only interval pointers $(f, [i, j), \hat{k})$ rather than materializing samples. Given M and chunk size c, Algorithm 1 constructs C_h such that for each mixture key $k \in \mathcal{K}$, the number of samples with component keys matching k approximates $M(k) \cdot c$. Users can set a mixture to be *strict*, requiring exact proportions, or *best-effort* (continue to generate chunks even if the mixture cannot be exactly fulfilled). The algorithm iterates through each $k \in \mathcal{K}$, identifies all matching component keys $\hat{k}$ in the `ChunkerIndex`, and extracts intervals until the target count is reached. This algorithm supports dynamic mixture, as the mixture can be changed between chunks.

For each key in the mixture, the algorithm keeps track of how many samples we still need to put into the chunk that is currently being generated (`remaining_sizes`). For each key in the mixture (line 6), it checks whether it matches a component key in the chunker index (lines 7-8). If we find a match, we try and obtain samples (ranges) from the `ChunkerIndex` for this component key (lines 9+). A call to obtain samples for a component key can return fewer samples than requested, e.g., if we are looking for JavaScript data and we need 5 samples, but we only have 3 JavaScript/MIT licensed samples, the according component key can only return 3 samples. Requesting n samples is implemented as requesting m intervals (from potentially multiple files) such that the overall number of samples in the returned list of intervals is $\leq n$. The lists of intervals per file are merged into the existing sorted list of intervals.

If, after traversing all component keys, we did not find sufficient samples for a key in the mixture, in strict mode, chunk generation fails. In best-effort mode, the algorithm redistributes any unfulfilled counts to the remaining mixture components proportionally to their original ratios. For example, if we need 100 JavaScript samples but only find 80, the remaining 20 samples would be proportionally distributed among other components. To avoid infinite loops, we only distribute samples to keys on which we were able to find any samples in the last iteration. This redistribution mechanism can be enabled or disabled, allowing users to either prioritize strict mixture fidelity (which may stop training when samples are exhausted) or training as long as possible with approximated proportions.

Implementation details. The *take samples* operation (line 9) is implemented using Python generators that yield ranges of samples and accept the number of samples needed as input through the generator's send mechanism. This hides the complexity of range management and allows for efficient, stateful iteration over available ranges while maintaining control over sample counts. There is one generator per component key that returns ranges containing N samples based on the `ChunkerIndex`, ensuring ranges are split such that excess data is never returned.

Determinism. Mixtera's implementation of this algorithm ensures determinism because (1) the keys are processed in a consistent order and (2) when multiple component keys match a mixture key, they are considered in a deterministic order based on a seeded shuffle of all possible keys. This ensures that identical queries with identical mixtures always produce identical chunks, which is important for debugging and reproducibility [18, 29, 51, 66, 83].

Sampling and randomness. Chunks are generated randomly, but not completely i.i.d.: (i) each chunk must contain the proportions specified by the mixture M, and (ii) all matching component keys are considered equal. Consider a mixture key k = {lang: JS} that matches component keys $\hat{k}_1$ = {lang: JS, license: MIT} and $\hat{k}_2$ = {lang: JS, license: Apache}. The order in which $\hat{k}_1$ and $\hat{k}_2$ are processed is determined by a seeded shuffle, ensuring determinism across runs while providing randomization across different matching keys. Once a component key is selected, samples are drawn from its intervals. Within each file, intervals are used for I/O efficiency, but the order in which files are processed is shuffled.

Importantly, if the sample count for mixture key k is satisfied by samples from $\hat{k}_1$, samples from $\hat{k}_2$ may never be selected for that chunk. This is not a sampling bias but rather reflects the mixture specification. By omitting the *license* property from k, the user declares all JavaScript samples equivalent regardless of license. To ensure unbiased sampling across all relevant dimensions, users should include all properties they care about in the mixture.

Mixture types. All mixture classes implemented in Mixtera share a common interface that converts their specifications into a mapping from `MixtureKeys` to sample counts per chunk, used by the chunk generation algorithm:

- Static Mixture: Users explicitly specify fixed proportions for different property combinations (Figure 4). This supports arbitrary properties and is not limited by, e.g., directory boundaries.
- Inferring Mixture: Automatically derives mixture proportions from the data distribution in the query result: This is useful when users want to maintain the natural distribution of properties.
- Hierarchical Mixture: An advanced static mixture that allows specifying nested property relationships. For example, users can define that 50 % of the data should be legal texts, and within that, 60 % should be in English and 40% in French. Mixtera automatically flattens this hierarchy into appropriate `MixtureKeys`.
- Mixture Schedule: A "meta mixture" that allows for temporal changes in mixture composition by defining a sequence of mixtures that activate at specific training steps. This enables curriculum learning with predefined schedules.
- Dynamic Mixture: Allows adaptation of mixture proportions during training based on feedback (e.g., loss) from the model. If an algorithm is already supported by Mixtera (e.g., ADO), it can be used directly.

Chunk distribution. In distributed training, it is important to guarantee that all nodes within the same data parallel group operate on the same input tensors. Mixtera's `ChunkDistributor` wraps around the chunk generation component, and hands out chunks correctly to the training nodes, i.e., the same chunks in the same order to nodes within the same group, and different chunks to nodes in different groups for data parallelism. To this end, the clients need to register at the server with their respective node and group identifiers. To avoid redundant serialization overhead, the distributor caches chunks in serialized form until all nodes in a group have received them.

5.2.3 Networking. We implement a TCP-based client-server protocol. The server uses Python's asyncio framework to handle multiple concurrent client connections. The protocol is message-based, with each message consisting of a task identifier followed by task-specific payload data. Tasks, for example, include the execution of a query or sending out a new chunk to a client. To handle network issues gracefully, the client implementation includes automatic reconnection with exponential backoff and configurable timeouts. Note that only small objects such as chunks and not actual training data is transferred via Mixtera, to avoid training bottlenecks.

5.3 Client-Side Reading

Mixtera's client-side abstractions provide a generator that, given a chunk from the server, yields the actual sample payloads $s \in \mathcal{S}$. This generator follows a two-level nested iteration pattern: an outer iteration over chunks and an inner iteration over samples within each chunk. The outer iteration hides the complexity of network transfer and chunk generation, while the inner iteration hides the complexity of going from pointers in the chunk to actual samples. The previous section discussed the outer step, and we now discuss this inner step, i.e., how, given a chunk, we yield sample payloads.

Sample granularity. A critical aspect of Mixtera's client-side reading is understanding the *sample unit.* As defined in Section 4.1, a sample s is the atomic unit of training data *as supplied by the user.* The sample granularity is inherent to the ingested dataset $\mathcal{S}$. Mixtera is a read-only layer and does not modify or redefine this granularity. In practice, sample granularity varies significantly, from books to partial sentences.

Varying granularities creates a challenge for mixture enforcement. If samples come in different lengths, enforcing mixture proportions at the *sample level* may not reflect the actual mixture at the *token level* seen during training. A single long sample from one domain may contribute orders of magnitude more tokens than a short sample from another domain. To address this, Mixtera provides three processing modes that offer different trade-offs between mixture guarantee granularity and sample utilization.

Processing modes. A chunk can be processed in three mixture processing modes with different trade-offs. The modes influence in what order samples are yielded, i.e., in what granularity the mixture is guaranteed, and whether string samples or tokenized sequences are yielded. All modes begin by instantiating one *active iterator* per property combination. These iterators traverse files and ranges for their respective properties in a randomized order while maintaining sequential reading within consecutive ranges for I/O efficiency.

Overall mixture mode. This mode processes active iterators in a randomized round-robin fashion until depletion. This ensures the mixture ratio is maintained at the chunk level.

Window mixture mode. This mode guarantees the mixture on a window smaller than the chunk size. Similarly to chunk generation, we determine how many samples per property we yield within a window. We then go through the properties in a randomized, round robin fashion until a window has been yielded, and start again. This mode can operate in best-effort (continues after mixture cannot be guaranteed) or strict mode (stops at the first window where the mixture cannot be maintained). In strict mode, the number of overall samples yielded from the chunk might be smaller than the chunk size.

Tokenized mixture mode. This mode addresses the sample granularity issue identified above by enforcing mixture ratios at the *token level* rather than the sample level. It wraps the active iterators with a *tokenizing iterator* that takes the incoming string samples, tokenizes them, and yields tokenized samples (integer lists) with the correct sequence length. By setting the window size equal to the chunk size we guarantee that each chunk yields at least one window of *tokenized samples*. This ensures that the mixture M is respected in terms of actual training tokens, regardless of the varying lengths of the underlying samples $s \in \mathcal{S}$.

While this mode ensures precise mixture ratios at the token level, it may result in partial utilization of longer samples. This is an inherent issue of unbalanced datasets, and while Mixtera provides flexibility to handle it, the best approach is to process the datasets such that samples are (roughly) of similar size.

File reading. The active iterators wrapped by the processing iterators shuffle the file order but maintain sequential reading within files. This acknowledges that formats like `jsonl` and `parquet` are optimized for sequential rather than random access, and enables us to linearly iterate through the sorted lists of ranges per file. The complexity of reading different file formats internally is hidden by abstractions for each format. Using the `xopen` library we support both compressed and uncompressed `jsonl`. We optimize the reading of `parquet` files by calculating and loading only the relevant the row groups, and build upon pyarrow's `parquet`-batched-reading implementation. WebDatasets is the only format supporting random access to samples, and we implement support using the `wids` library. The format also gives us the option to store text-image pairs, as it can contain different modalities. To mitigate latency from initial file operations that we observed on distributed filesystems, Mixtera employs a prefetching iterator that uses background threads to hide file opening latency.

Table 2: Model configurations.

	Hid. Dim.	Interm. Dim.	KV-Hds.	Q-Hds.	Layers	RoPE-θ
162M	768	2 048	12	12	12	10 000
1.6B	2 048	5 464	16	16	24	10 000
3.6B	3 072	8 192	8	24	28	500 000

Determinism. All random operations are seeded based on the current chunk, ensuring identical behavior across nodes. Combined with the server-side chunk generation and distribution, this guarantees that *all clients within a data parallel group yield exactly the same samples in exactly the same order*. This property is crucial for both reproducibility across runs and correctness in distributed training. We validated this using a suite of integration tests as well as the dataloader verification test provided by nanotron.

Dataset abstractions. Training frameworks typically require specific dataset interfaces that support `multiprocessing` with worker processes. Besides a general-purpose interface, Mixtera offers a class extending torch's `IterableDataset`, and a class compatible with the Hugging Face API. Each worker process at each node operates on its own chunk.

5.4 Framework Integration

Mixtera integrates into the training framework for checkpointing and transferring model feedback (e.g., per-domain loss).

Checkpointing. Mixtera's API offers a function to be called on checkpoint. To restore from a checkpoint, we need to know (i) which chunks have been handed out to which nodes and (ii) which data loader worker processes have yielded how many samples for each node. Mixtera implements (i) using the `ChunkDistributor`, which caches the query and the current state on checkpoint and can restore the in-memory state of the iterators for chunk generation based on this information. For (ii) the `MixteraTorchDataset` uses a shared memory segment to share with the main training process the status of the data loader workers. When we restore a checkpoint, we restore the state at the server, hand out the last chunks to each worker that they were working on, and then at the workers discard the previously yielded samples.

Training feedback. Dynamic mixing algorithms require a loss per property domain. Mixtera offers a simple function to forward this information to the server. Users still need to adjust the training framework. The loss implementation needs to be adjusted s.t. it is not immediately reduced but stored per domain. The per-domain losses along all training nodes need to be synchronized, e.g., via all-reduce, before passing it to Mixtera.

6 Evaluation

We evaluate Mixtera to answer the following questions:

(1) How can we integrate dynamic mixing algorithms into Mixtera and what role do mixtures play for model accuracy?
(2) How does Mixtera's throughput compare to other data loaders and how well does it scale?

We explore the first question in a dynamic mixture case study on LLMs (Section 6.1) and a static mixture case study for VLMs

Table 3: Task performance and perplexities across models and mixtures. ↑/↓ indicate higher/lower is better.

Model	Mixture	HellaSwag ↑	WinoGrande ↑	ARC-E ↑	ARC-C ↑	Lambada (OAI) ↑	OpenBookQA ↑	PIQA ↑	SlimP. Perp. ↓	Pile Perp. ↓
1.6B	ADO	**0.411**	**0.577**	**0.577**	**0.262**	**0.603**	**0.236**	**0.707**	**25.20**	**27.79**
	Default	0.380	0.549	0.543	0.243	0.551	0.218	0.695	26.55	30.72
	Natural	0.383	0.556	0.559	0.247	0.562	0.214	0.698	26.04	29.94
3.6B	ADO	**0.449**	**0.626**	**0.601**	**0.276**	**0.624**	**0.244**	**0.732**	22.58	22.26
	Default	0.415	0.597	0.579	0.255	0.592	0.210	0.706	22.09	**22.02**
	Natural	0.419	0.586	0.593	**0.276**	0.598	0.206	0.723	**21.99**	22.53

(Section 6.2). We explore the second question with throughput benchmarks (Section 6.3).

Setup. We run experiments on HPE Cray Supercomputing EX254n blades, each hosting two Quad GH200 nodes. Each node contains 4 interconnected groups of a Grace CPU with 72 cores, 128 GB DRAM, and a H100 Hopper GPU with 96 GB of HBM. The nodes are connected using a 200 Gb/s HPE Slingshot interconnect. The machines run Ubuntu Server 24.04 LTS with kernel 5.14.21. We build on the NVIDIA NGC 25.01 container with Python 3.12, a nightly build of PyTorch 2.7, NVIDIA driver 550.54.15, and CUDA 12.8. We add support for Mixtera and other data loaders on our fork of Torchtitan (commit `ae4e402`)[2] [34]. Torchtitan is part of the PyTorch ecosystem and straightforward to set up. We use Llama3-like models with configurations in Table 2. The 162M and 1.6B models are based on Jiang et al. [28], while 3.6B follows Meta's Llama 3.2 model. They do not have the same parameter count as torchtitan does not tie the weight embeddings. Our training and benchmarking data is based on The Pile [20], a frequently-used data collection used for exploring data mixing [3, 28, 71]. We split long samples with more than 1 500 words, with max. 20 k samples per file.

6.1 Dynamic Mixing using ADO

We demonstrate how to implement dynamic mixing algorithms in Mixtera and their impact on model performance, taking the ADO algorithm (Section 2.1.1) as an example.

Training setup. We test the 1.6B model from Jiang et al. [28] with the `EleutherAI/GPT-NeoX-20B` tokenizer and a 3.6B model (Table 2) following `Llama-3.2-3B` from Meta, including its tokenizer. We omit the results from the 162M model for brevity. We use a sequence length of 2048. For ADO, we follow the codebase and discard the first 500 steps for fitting the scaling laws, start with fitting them at step 1 000, and then re-fit the laws every 1 000 steps with a subsampling frequency of 10. We also follow the codebase and "use the same step size" for all domains, i.e., instead of using the count of how often a domain has been sampled to fit the scaling laws like in the paper, we average the total sample counts evenly across all domains. We train using non-strict token-level mixtures for 30 000 steps, using a learning rate of 0.001 with a linear warmup for 500 steps and linear cooldown for 3 000 steps, and the AdamW optimizer. For 1.6B, we use 64 GPUs with a microbatch size of 32, and for 3.6B we use 128 GPUs with a microbatch size of 16, resulting in a global batch size of 2048 and total 125 B tokens. As mixtures, we test the default weights as in DoReMi [71], the natural weights as in Jiang et al. [28], and ADO initialized with the natural weights.

[2]Available at https://github.com/eth-easl/torchtitan-mixtera.

Evaluation metrics. We follow Jiang et al. [28] and report both downstream task performance as well as perplexity. For downstream tasks, we report performance on HellaSwag [78], WinoGrande [56], ARC-Easy and ARC-Challenge [16], Lambada OpenAI [48], OpenBookQA [41], and PIQA [7]. For perplexity, we report the average unweighted token perplexity on (i) the validation set of The Pile [20], and on (ii) SlimPajama [61] as a dataset we did not train on. We collect all metrics using EleutherAI's lm-eval-harness [21], and use the unnormalized accuracy.

ADO algorithm performance overview. Table 3 shows the performance of all models and mixtures for the final checkpoint after learning rate cooldown. We mark in bold the best value within a model/step group. Generally, ADO beats the static mixtures across all downstream benchmarks. On 1.6B, it also has the best (lowest) perplexity on both SlimPajama and The Pile, while on 3.6B, the static mixtures have slightly lower perplexity. This shows that ADO leads to better performance for downstream tasks across scales, and demonstrates the usage of a dynamic mixing algorithm in Mixtera.

Jiang et al. [28] report that on the 1.6B model, ADO sometimes performs worse than static mixtures, which we do not confirm. Since ADO's official repository is tightly coupled with the training framework, even after corresponding with the authors, we were not able to identify the root cause of this, partly also because their code is bound to training on specific cloud instances. Mixtera decouples the training framework from the mixing algorithm, which helps developers port existing algorithms to their setups (Section 3).

Performance over time. To demonstrate how different tasks behave over the training, we show the performance of the 1.6B model on HellaSwag, OpenBookQA, and ARC-Easy for all training checkpoints in Figure 5. Every benchmark exhibits different behavior. For HellaSwag, ADO consistently increases its margin over the static mixtures. For OpenBookQA, ADO performs similarly to the natural mix in the intermediate checkpoints, and benefits a lot during the learning rate cooldown. For ARC-Easy, the mixtures perform similarly, with ADO having a small edge. This motivates future research on data mixing using Mixtera–for example, we might be able to use intermediate evaluations instead of loss to dynamically adjust the mixture.

Mixture over time. We showcase the mixture over time for the six largest domains in Figure 6. We show the mixture obtained on the 1.6B model, but unlike Jiang et al. [28], we do not observe that other models/tokenizers lead to largely different mixtures. In all cases, the weight of GitHub and ArXiv rapidly decreases. Notably, the more parameters, the higher the weight of Books3, and the lower the weight of PubMed Central. On the 3.6B model, OpenWebText2's

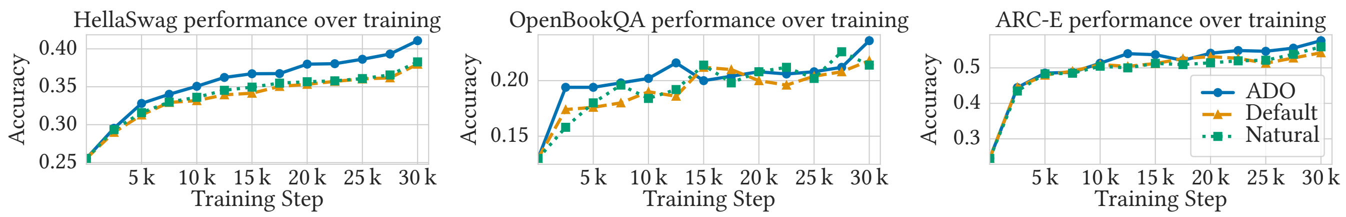


Figure 5: Performance of the 1.6B model on HellaSwag, OpenBookQA, and ARC-Easy, measured every 2 500 steps.

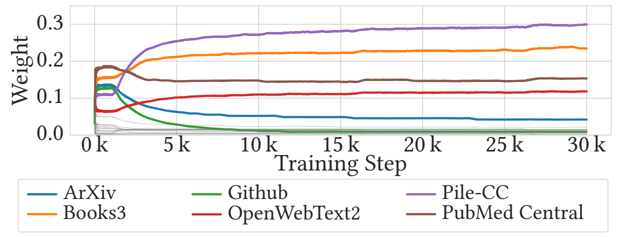


Figure 6: Mixture for the 1.6B model for the 6 largest domains.

weight surpasses PubMed Central's weight before step 5 000, while for the 1.6B model, they only slowly approach.

Mixtera implementation. We implement ADO in Mixtera in ca. 800 LOC. The class only handles the core algorithm, while Mixtera hides the complexity of the actual mixing from the algorithm implementation. We implement it from scratch as the original implementation of ADO is fully tied to their training framework and data setup. The original ADO implementation updates the current mixture π after every step and samples the next batch based on this distribution. This is a small discrepancy to Mixtera, which can only use a new mixture when generating a new chunk. Each chunk may then yield several batches of data with the same mixture. We still send the per-domain losses on every training step at the client to update ADO's internal state at the server. Whenever a new chunk is generated, the current mixture π from ADO is queried (Algorithm 1), and the server generates a chunk according to that mixture. As we find in the experiments, this slight slack does not harm performance, due to the stochastic nature of sampling.

In order to use ADO, at the training nodes, the only change needed is the implementation of a per-domain loss. For this, the loss function (e.g., cross-entropy) needs to be called without reduction, which gives a loss *per token*. As Mixtera provides which token belongs to which domain, we can aggregate the losses *per domain*. We then perform an all-reduce operation across all training nodes to get the global per-domain losses and send this to the server.

During development, we switched from nanotron [26] to torchtitan [34]. Notably, as Mixtera is agnostic to the training framework, no changes in Mixtera were required. This showcases the benefit of having a system like Mixtera that decouples the mixing from the training framework.

Takeaways. Dynamic mixtures can improve model accuracy. ADO scales beyond the 1.6B model tested in the original paper, and beats the default weights on all benchmarks. Our experiments also demonstrate that a synchronous algorithm, which uses a new mixture at each training step, adapts to Mixtera's chunking system.

Table 4: VLM scores for hand-picked mixtures.

Model	SQA-IMG	TxtVQA	GQA	MME	MMMU	POPE
Jia et al. [27]	64.0	49.6	58.6	1256.5	28.3	86.3
Infer. Mix.	55.88	37.92	54.63	1238.76	**29.4**	**86.6**
Mix. #252	63.01	43.87	56.14	**1268.05**	**30.4**	85.7
Mix. #107	58.50	45.18	58.36	**1283.62**	**30.1**	85.54
Mix. #155	57.14	42.37	57.14	**1290.06**	**29.3**	**86.63**

6.2 Multimodal LLaVA Finetuning

To showcase Mixtera's multimodal capabilities, we evaluate the impact of static mixtures for finetuning a LLaVA-style model [36]. We are not aware of prior work on the impact of data mixtures on VLMs. The LLaVA framework trains an adapter between a pre-trained image encoder and a pre-trained LLM, and then fine-tunes the adapter and LLM on visual instruction-following data.

Training setup. We base our training setup on the TinyLlaVA Factory codebase by Jia et al. [27]. We rely on a recipe from the Factory and use `google/siglip-so400m-patch14-384` [2] as the vision encoder, `TinyLlama/TinyLlama-1.1B-Chat-v1.0` [79] as the LLM, and a 2-layer MLP as the adapter [64]. We train all models on one GH200 node with 4 data parallel GPUs. For pre-training, we use a global batch size of 512 with a learning rate of 0.001, and for finetuning use a global batch size of 128 with a learning rate of 0.00002. We use a chunk size of 256, a cosine learning rate scheduler and the Adam optimizer. We follow Liu et al. [36, 63] and pre-train the adapter on a 558 k subset of the LAION-CC-SBU dataset with BLIP captions. For finetuning, we follow the TinyLLava Factory [67] "LLaVA dataset" and use 665 k samples from six datasets (COCO [35], GQA [25], OCR-VQA [42], TextVQA [60], and VisualGenome (VG) [30], and LLaVA's text-only SFT annotations [36]). We pre-train the adapter once, and then vary the proportions of the finetuning datasets. We randomly generate 256 mixtures for finetuning. Since the number of datapoints in comparison to LLM training is small, we use a best-effort mixture and ensure we go through all samples exactly once. All models see the same data, but in a different order.

Benchmarks. We evaluate the models on the GQA [25], SQA-IMG [38], TextVQA [60], POPE [32], MME [19], and MMMU [76] benchmarks. We collect all metrics using TinyLLaVA Factory.

Results. In Table 4, we show the results reported by Jia et al. [27], the results we obtain using the inferring mixture (Section 5.2.2), and three mixtures out of the generated mixtures that perform well. While the inferring mixture does not achieve their reported results, this could be either due to different data dynamics in their training, or due to a different evaluation setup we cannot reproduce as their model weights are not public. Nevertheless, in particular on MME/MMMU/POPE, the mixtures outperform the baseline. Mix. #252 is weighted towards TextVQA (29.1 %) and OCR-VQA

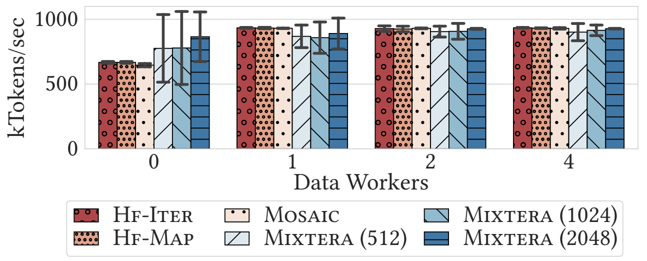


Figure 7: Data loader throughput depending on the number of workers. The number in brackets indicates the chunk size.

(19.5 %), with equal proportions of COCO (21.9%) and VG (21.9 %); GQA (4.0 %) and SFT annotations (3.6 %) contribute minimally. Mix. #107 places a large emphasis on SFT annotations (28.9 %) and VG (27.4 %), followed by COCO (25.5 %); GQA (5.3 %) and TextVQA (5.1 %) have lower representation, while OCR-VQA (7.8 %) remains a minor component. Mix. #155 prioritizes OCR-VQA (30.3 %) and TextVQA (25.7 %), and SFT annotations (22.7 %); VG (13.2 %) and COCO (6.9 %) are underrepresented, while GQA (1.2 %) is least utilized. Overall, despite seeing the same samples globally, the mixtures play a big role for model accuracy.

6.3 Throughput Benchmarks

We now focus on throughput and compare MIXTERA with other data loaders across various configurations. The goal is avoiding data stalls, i.e., training throughput should not be reduced because the GPU is waiting for data [10, 15, 22, 31, 43, 44, 55, 81]. We want to show that MIXTERA enables on-the-fly data streaming with dynamic mixing without lowering throughput in comparison to other data loaders. We measure throughput in tokens per second. Note that *smaller models* and *more data parallelism* increase the pressure on the data loader, while larger models reduce the pressure as the training computation takes longer. If a data loader can sustain training small models at scale on a high-end platform like the GH200, its performance is sufficient for other scenarios as well.

We run MIXTERA with chunk sizes of 512, 1024, and 2048. We compare to well-known state-of-the-art data loaders, i.e., the iterable `HuggingFaceDataset` by Torchtitan (HF-ITER), a mapped version (HF-MAP), and the Mosaic StreamingDataset [65] (MOSAIC). The difference between HF-ITER and HF-MAP is that similar to MIXTERA, HF-ITER loads and tokenizes the data on the fly, while HF-MAP preprocesses all data, including tokenization. Note that none of these data loaders support dynamic mixing (Table 1), they just read files from front to back. We evaluate throughput on the 162M model since larger models only lead to lower throughput, as discussed above. We always use FSDP since Torchtitan only enables `bfloat16` training if sharding is used. We activate compilation, disable activation checkpointing, and use fused AdamW. We measure throughput for 30 steps, discarding the first step. We repeat all measurements three times, i.e., in total we have 3x30 steps. We use the Hugging Face `EleutherAI/gpt-neox-20b` tokenizer for all data loaders. We store all data on an SSD-backed Lustre DFS.

Single-node. We train on a single GH200 node with 4 GPUs. We use 2 data parallel replicates and shards. In Figure 7 we show the throughput for the data loaders depending on the number of data workers, i.e., data loading processes from the `torch.DataLoader`.

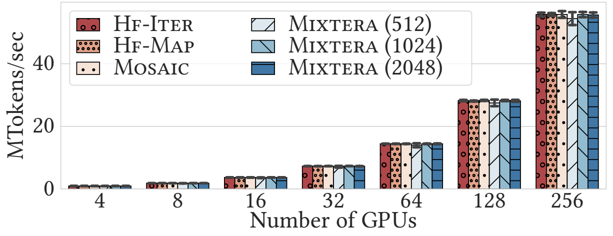


Figure 8: Data loader throughput when increasing the number of data parallel nodes.

We test up to 16 workers, where 0 workers indicate that data is loaded in the same process as the main training loop. The plot only show results up to 4 workers as throughput does not increase further. Without data workers, MIXTERA has the highest average throughput; with one worker, the other data loaders reach their peak performance.

Overall, MIXTERA provides similar throughput to the baselines *while having a much richer feature set*, e.g., dynamic mixing. Notably, MIXTERA has higher throughput variance than other loaders for lower number of workers. This is due to the random access into the files. For every sample, it needs to open the file, seek to the correct position, and load the data, instead of bulk-transferring all the data as the other data loaders can do. This leads to the higher variance in throughput indicated by the error bars, which overall leads to slightly lower averages. With a higher number of workers and larger chunk sizes, the variance decreases. When using 4+ workers, MIXTERA's throughput matches the throughput of the baselines. Additionally, increasing the chunk size also helps, and for the 0 worker case, MIXTERA with a chunk size of 1024 or 2048 even has a higher average throughput.

This benchmark setup is quite extreme, as we train a very small 162M model on an extremely fast GPU. For larger, state-of-the-art model sizes, the throughput differences disappear, as computation time spent within the model forward and backward passes increases. MIXTERA's client-side overhead is minimal; the primary operations (chunk parsing and data fetching) introduce negligible overhead.

Scaling out. We investigate how the data loaders scale for larger training jobs with higher data parallelism. We scale up to 64 GH200 nodes with a total of 256 data parallel GPUs. We find that using a maximum number of 16 replication GPUs works best. For 4, 8, and 16 GPUs, we use half the GPUs as replication, and shard across the rest. Figure 8 shows the results with 8 data workers. All data loaders scale linearly. With more GPUs, the throughput variance increases a bit for all systems. We attribute this to the random assignment of nodes in the cluster by the Slurm scheduler across the 3 repetitions. We do not test pipeline or tensor parallelism as (i) this is not necessary for the 162M model (and a larger model would only stress the data loaders less), (ii) the data loaders besides MIXTERA do not easily support 3D parallelism, (iii) increasing data parallelism increases the load on the system more. This demonstrates the scalability of MIXTERA's single-controller design and implementation, as well as the efficiency of its chunk generation, enabling it to effectively supply chunks to all clients even at scale.

File formats. All previous benchmarks use uncompressed `jsonl` data. We test compressed `jsonl` (`jsonl.zst`), `parquet`, and the

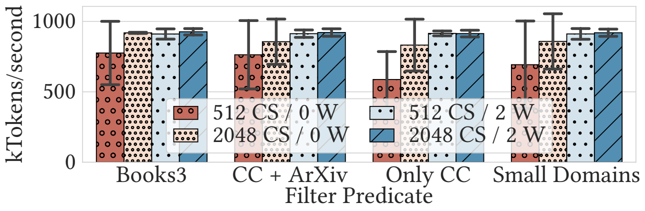


Figure 9: Throughput depending on the filtering predicate with 0 and 2 workers.

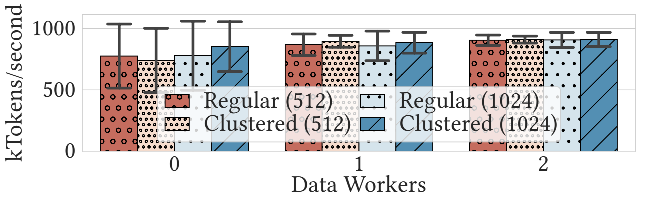


Figure 10: Throughput on clustered vs. regular data.

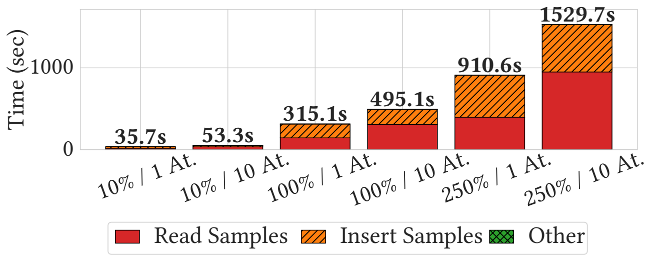


Figure 11: Ingestion time breakdown on 10 %, 100 %, and 250 % versions of The Pile with 1 and 10 attributes (3 run average).

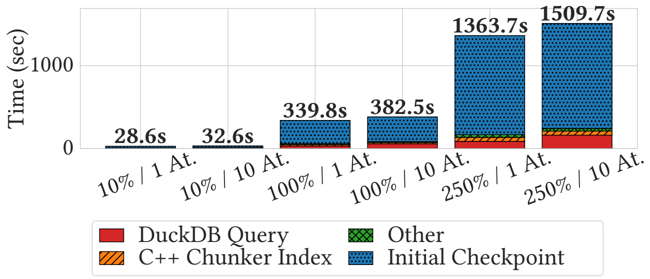


Figure 12: Execution time breakdown on 10 %, 100 %, and 250 % versions of The Pile with 1 and 10 attributes (3 run average).

`webdatasets` format in the data loaders that support them. Notably, only MIXTERA supports all of these formats. We find that the underlying file format does not impact the training throughput and therefore omit a plot. This observation holds across different numbers of data workers, where we also observe minimal performance variation among the data loaders.

Predicates. The previous benchmarks select all domains within The Pile. To show how a filtering predicate impacts throughput, in Figure 9, we show the throughput for chunks sizes 512/2048, and 0/2 data workers, across four representative predicates: Books3 only (large samples), CommonCrawl (CC) and ArXiv (frequent domains with small and large samples), CC only (frequent but small samples), and a set of infrequent domains (`Enron Emails`, `NIH ExPorter`, `PhilPapers`, `EuroParl`, `USPTO`).

With 0 data workers, throughput varies with the predicate; more workers stabilize throughput. For chunk size 512, Only CC achieves ~587 kTokens/s, while Books3 reaches ~776 kTokens/s. These differences stem from different file access patterns. Because The Pile is shuffled, large domains are more likely to appear in clusters, reducing repeated file scans. Domains also differ in sample sizes: large text samples yield multiple training examples from a single read, lowering total data transfer, as seen with Books3. With two workers, MIXTERA masks these effects and throughput is uniform.

Clustering. If data is clustered, MIXTERA can generate longer intervals and thereby reduce the amount of data transfers. Figure 10 compares the throughput of the regular benchmarking data with a clustered version that sequentially writes out each domain's data. For a small number of workers, we generally observe an increase in avg. throughput and a slight decrease of variance. As before, using more workers hides these effects.

6.4 Data Ingestion

Before executing queries, the user must ingest metadata (c.f. Section 5.1). During ingestion, MIXTERA reads all samples, extracts their metadata, and inserts it into the underlying metadata database. This cost is paid *only once*; thereafter, arbitrarily many queries can be executed. In Figure 11 we report time spent reading the samples (scanning all files) and inserting metadata. We benchmark this for varying dataset sizes (10 %, 100 %, and 250 % of The Pile) and number of attributes (randomly generated to avoid correlation). Other operations (e.g., parser setup) have negligible overhead.

The timings are obtained using MIXTERA's optimized default settings: multithreaded file scanning (MT) and an insertion chunk size of 2,000. Disabling MT increases the ingestion time from 315 to around 3,000 seconds for the standard 100 % / 1 attribute case. With MT enabled, decreasing the chunk size to 500 increases ingestion time from 315 s to 340 s. Chunk sizes larger than 2,000 yield only marginal benefit and increase the risk of out-of-memory errors.

6.5 Query Execution Latency Breakdown

When a query is executed in MIXTERA, the main steps are querying the populated ODC (DuckDB), building the `ChunkerIndex`, and, optionally, writing the first checkpoint. In Figure 12, we give a time breakdown for varying dataset sizes (10 %, 100 %, and 250 % of The Pile) and number of properties (randomly generated to avoid correlation), following Section 6.4. For the 100 % / 1 attribute case, DuckDB takes 32 s, and preparing the index takes 16 s due to our C++ implementation. Persisting the initial state checkpoint of the query dominates the runtime with 282 s. This is because we have to serialize nested Python dictionaries, which is slow despite engineering optimizations. Importantly, after this initial checkpoint, future checkpoints at the server can be stored *within milliseconds*, since we do not have to serialize the index again. If no MIXTERA checkpoints are needed, the serialization can be skipped.

Other systems. The streaming HF-ITER data loader can basically start streaming data immediately, the HF-MAP data loader, the default in nanotron, loads and tokenizes the data first, taking 2 h 51 min for The Pile 100 %. However, to use streaming data loaders such as HF-ITER or MOSAIC, in many scenarios, users would also need to run more offline preprocessing, e.g., to perform static filtering, or reshuffling the data if we want to mix on a different property. MIXTERA avoids this offline preprocessing completely.

7 Conclusion and Future Work

Understanding how data mixture recipes affect model quality is an active and critical area of ML research. We design Mixtera to enable researchers and model developers to easily and quickly train models with a variety of mixtures across arbitrary data properties and dynamically vary mixtures during training. Mixtera is a declarative data plane for foundation model training that is training framework-agnostic. We demonstrate Mixtera's throughput and scalability, as well as the impact of mixtures on model quality for both LLMs and VLMs. The system lays the foundation for implementing additional features, such as data lineage tracking for model training, as it has a global view of the data, and enables future research on compute-intensive modalities such as video [74] and audio [80].

Acknowledgments

We thank Beste Güney for her contributions to Mixtera's codebase. We thank Antoni-Joan Solergibert i Llaquet, Imanol Schlag, Steven Hand, Martin Jaggi, Antoine Bosselut, Alexander Hägele, Loubna Ben Allal, Quentin de Laroussilhe, Paul Barham, Yiding Jiang, Theodoros Rekatsinas, Gustavo Alonso, and Foteini Strati for helpful discussions. This work was supported as part of the Swiss AI Initiative by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID a09 on Alps. Maximilian Böther is supported by the Swiss National Science Foundation (project number 200021_204620).

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015). doi:10.14778/2824032.2824076

[2] Ibrahim M. Alabdulmohsin, Xiaohua Zhai, Alexander Kolesnikov, and Lucas Beyer. 2023. Getting ViT in Shape: Scaling Laws for Compute-Optimal Model Design. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[3] Alon Albalak, Liangming Pan, Colin Raffel, and William Yang Wang. 2023. Efficient Online Data Mixing For Language Model Pre-Training. *arXiv Preprint* (2023). doi:10.48550/arXiv.2312.02406

[4] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. 2025. SmolLM2: When Smol Goes Big – Data-Centric Training of a Small Language Model. *arXiv preprint* (2025). doi:10.48550/arXiv.2502.02737

[5] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. 2022. Pathways: Asynchronous Distributed Dataflow for ML. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*.

[6] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *Comput. Surveys* 52, 4 (2019), 1–43. doi:10.1145/3320060

[7] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2020. PIQA: Reasoning about Physical Commonsense in Natural Language. *Proceedings of the Conference on Artificial Intelligence (AAAI)*. doi:10.1609/aaai.v34i05.6239

[8] Jan-Micha Bodensohn, Ulf Brackmann, Liane Vogel, Anupam Sanghi, and Carsten Binnig. 2025. Unveiling Challenges for LLMs in Enterprise Data Engineering. *arXiv preprint* (2025). doi:10.48550/arXiv.2504.10950

[9] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. 2022. On the Opportunities and Risks of Foundation Models. In *arXiv preprint*. doi:10.48550/arXiv.2108.07258

[10] Maximilian Böther, Ties Robroek, Viktor Gsteiger, Xianzhe Ma, Pınar Tözün, and Ana Klimovic. 2025. Modyn: Data-Centric Machine Learning Pipeline Orchestration. In *Proceedings of the Conference on Management of Data (SIGMOD)*. doi:10.1145/3709705

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[12] Daoyuan Chen, Yilun Huang, Zhijian Ma, Hesen Chen, Xuchen Pan, Ce Ge, Dawei Gao, Yuexiang Xie, Zhaoyang Liu, Jinyang Gao, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. Data-Juicer: A One-Stop Data Processing System for Large Language Models. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. doi:10.1145/3626246.3653385

[13] Mayee F. Chen, Michael Y. Hu, Nicholas Lourie, Kyunghyun Cho, and Christopher Ré. 2024. Aioli: A Unified Optimization Framework for Language Model Data Mixing. In *Proceedings of the International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2411.05735

[14] Mayee F. Chen, Nicholas Roberts, Kush Bhatia, Jue Wang, Ce Zhang, Frederic Sala, and Christopher Ré. 2023. Skill-it! A Data-Driven Skills Framework for Understanding and Training Language Models. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*. doi:10.48550/arXiv.2307.14430

[15] Zihao Chen, Chenyang Zhang, Chen Xu, Zhao Zhang, Jiaqiang Wang, Weining Qian, and Aoying Zhou. 2025. Scheduling Data Processing Pipelines for Incremental Training on MLP-based Recommendation Models. In *Companion of the International Conference on Management of Data (SIGMOD)*. doi:10.1145/3722212.3724454

[16] 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. *arXiv preprint* (2018). doi:10.48550/arXiv.1803.05457

[17] Competition and Markets Authority. 2013. *AI Foundation Models: Initial Report.* Technical Report. UK Government Agency.

[18] A. Feder Cooper, Jonathan Frankle, and Christopher De Sa. 2022. Non-Determinism and the Lawlessness of Machine Learning Code. In *Proceedings of the Symposium on Computer Science and Law (CSLAW)*. doi:10.1145/3511265.3550446

[19] Chaoyou Fu, Peixian Chen, Yunhang Shen, Yulei Qin, Mengdan Zhang, Xu Lin, Jinrui Yang, Xiawu Zheng, Ke Li, Xing Sun, Yunsheng Wu, and Rongrong Ji. 2024. MME: A Comprehensive Evaluation Benchmark for Multimodal Large Language Models. *arXiv preprint* (2024). doi:10.48550/ARXIV.2306.13394

[20] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint* (2020). doi:10.48550/arXiv.2101.00027

[21] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 2023. A framework for few-shot language model evaluation. doi:10.5281/zenodo.10256836

[22] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning Input Data Processing as a Service. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[23] Torsten Hoefler, Tommaso Bonoto, Daniele De Sensi, Salvatore Di Girolamo, Shigang Li, Marco Heddes, Deepak Goel, Miguel Castro, and Steve Scott. 2024. HammingMesh: A Network Topology for Large-Scale Deep Learning. *Commun. ACM* 67, 12 (2024), 97–105. doi:10.1145/3623490

[24] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[25] Drew A. Hudson and Christopher D. Manning. 2019. GQA: A New Dataset for Real-World Visual Reasoning and Compositional Question Answering. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2019.00686

[26] HuggingFace. 2025. *Nanotron: Pretraining models made easy.* https://github.com/huggingface/nanotron

[27] Junlong Jia, Ying Hu, Xi Weng, Yiming Shi, Miao Li, Xingjian Zhang, Baichuan Zhou, Ziyu Liu, Jie Luo, Lei Huang, and Ji Wu. 2024. TinyLLaVA Factory: A Modularized Codebase for Small-scale Large Multimodal Models. *arXiv preprint* (2024). doi:10.48550/arXiv.2405.11788

[28] Yiding Jiang, Allan Zhou, Zhili Feng, Sadhika Malladi, and J. Zico Kolter. 2024. Adaptive Data Optimization: Dynamic Sample Selection with Scaling Laws. In *Proceedings of the International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2410.11820

[29] Siddharth Karamcheti, Laurel Orr, Jason Bolton, Tianyi Zhang, Karan Goel, Avanika Narayan, Rishi Bommasani, Deepak Narayanan, Tatsunori Hashimoto, Dan Jurafsky, Christopher D. Manning, Christopher Potts, Christopher Ré, and Percy Liang. 2021. Mistral - A Journey towards Reproducible Language Model Training.

[30] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A. Shamma, Michael S. Bernstein, and Li Fei-Fei. 2017. Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations. *International Journal of Computer Vision* 123, 1 (2017). doi:10.1007/s11263-016-0981-7

[31] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. 2022. Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*.

[32] Yifan Li, Yifan Du, Kun Zhou, Jinpeng Wang, Xin Zhao, and Ji-Rong Wen. 2023. Evaluating Object Hallucination in Large Vision-Language Models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. doi:10.18653/v1/2023.emnlp-main.20

[33] Zeman Li, Yuan Deng, Peilin Zhong, Meisam Razaviyayn, and Vahab Mirrokni. 2025. PiKE: Adaptive Data Mixing for Multi-Task Learning Under Low Gradient Conflicts. *arXiv preprint* (2025). doi:10.48550/arXiv.2502.06244

[34] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. 2024. TorchTitan: One-stop PyTorch native solution for production ready LLM pre-training. *arXiv preprint* (2024). doi:10.48550/arXiv.2410.06511

[35] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *Proceedings of the European Conference on Computer Vision (ECCV)*. doi:10.1007/978-3-319-10602-1_48

[36] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. 2024. Improved Baselines with Visual Instruction Tuning. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr52733.2024.02484

[37] Shayne Longpre, Gregory Yauney, Emily Reif, Katherine Lee, Adam Roberts, Barret Zoph, Denny Zhou, Jason Wei, Kevin Robinson, David Mimno, and Daphne Ippolito. 2024. A Pretrainer's Guide to Training Data: Measuring the Effects of Data Age, Domain Coverage, Quality, & Toxicity. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. doi:10.18653/v1/2024.naacl-long.179

[38] Pan Lu, Swaroop Mishra, Tony Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. 2022. Learn to Explain: Multimodal Reasoning via Thought Chains for Science Question Answering. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[39] Meta. 2024. The Llama 3 Herd of Models. *arXiv preprint* (2024). doi:10.48550/arXiv.2407.21783

[40] Meta. 2024. Llama 3.3 Model Card. https://github.com/meta-llama/llama-models/blob/main/models/llama3_3/MODEL_CARD.md. Accessed: 2024-12-18.

[41] 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. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. doi:10.18653/v1/d18-1260

[42] Anand Mishra, Shashank Shekhar, Ajeet Kumar Singh, and Anirban Chakraborty. 2019. OCR-VQA: Visual Question Answering by Reading Text in Images. In *Proceedings of the International Conference on Document Analysis and Recognition (ICDAR)*. doi:10.1109/icdar.2019.00156

[43] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and mitigating data stalls in DNN training. *Proceedings of the VLDB Endowment* 14, 5 (2021). doi:10.14778/3446095.3446100

[44] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. tf.data: a machine learning data processing framework. *Proceedings of the VLDB Endowment* 14, 12 (2021). doi:10.14778/3476311.3476374

[45] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/3341301.3359646

[46] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1145/3458817.3476209

[47] OpenAI. 2024. GPT-4 Technical Report. In *arXiv preprint*. doi:10.48550/arXiv.2303.08774

[48] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernandez. 2016. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. doi:10.18653/v1/p16-1144

[49] Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. 2024. The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale. *arXiv preprint* (2024). doi:10.48550/arXiv.2406.17557

[50] Guilherme Penedo, Hynek Kydlíček, Alessandro Cappelli, Mario Sasko, and Thomas Wolf. 2024. *DataTrove: large scale data processing.* https://github.com/huggingface/datatrove

[51] Shangshu Qian, Hung Viet Pham, Thibaud Lutellier, Zeou Hu, Jungwon Kim, Lin Tan, Yaoliang Yu, Jiahao Chen, and Sameena Shah. 2021. Are My Deep Learning Systems Fair? An Empirical Study of Fixed-Seed Training. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[52] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. doi:10.1145/3299869.3320212

[53] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. In *Self-hosted preprint*.

[54] Aquia Richburg and Marine Carpuat. 2024. How Multilingual Are Large Language Models Fine-Tuned for Translation? *arXiv preprint* (2024). doi:10.48550/arXiv.2405.20512

[55] Ties Robroek, Neil Kim Nielsen, and Pınar Tözün. 2026. TensorSocket: Shared Data Loading for Deep Learning Training. In *Proceedings of the Conference on Management of Data (SIGMOD)*. doi:10.1145/3749185

[56] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. WinoGrande: An Adversarial Winograd Schema Challenge at Scale. *Commun. ACM* 64, 9 (2021). doi:10.1145/3474381

[57] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[58] Zhiqiang Shen, Tianhua Tao, Liqun Ma, Willie Neiswanger, Zhengzhong Liu, Hongyi Wang, Bowen Tan, Joel Hestness, Natalia Vassilieva, Daria Soboleva, and Eric Xing. 2024. SlimPajama-DC: Understanding Data Combinations for LLM Training. *arXiv preprint* (2024). 10.48550/arXiv.2309.10818

[59] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint* (2019). doi:10.48550/arXiv.1909.08053

[60] Amanpreet Singh, Vivek Natarajan, Meet Shah, Yu Jiang, Xinlei Chen, Dhruv Batra, Devi Parikh, and Marcus Rohrbach. 2019. Towards VQA Models That Can Read. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2019.00851

[61] Daria Soboleva, Faisal Al-Khateeb, Robert Myers, Jacob R Steeves, Joel Hestness, and Nolan Dey. 2023. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama. https://huggingface.co/datasets/cerebras/SlimPajama-627B

[62] Luca Soldaini, Rodney Kinney, Akshita Bhagia, Dustin Schwenk, David Atkinson, Russell Authur, Ben Bogin, Khyathi Chandu, Jennifer Dumas, Yanai Elazar, Valentin Hofmann, Ananya Harsh Jha, Sachin Kumar, Li Lucy, Xinxi Lyu, Nathan Lambert, Ian Magnusson, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Abhilasha Ravichander, Kyle Richardson, Zejiang Shen, Emma Strubell, Nishant Subramani, Oyvind Tafjord, Pete Walsh,

Luke Zettlemoyer, Noah A. Smith, Hannaneh Hajishirzi, Iz Beltagy, Dirk Groeneveld, Jesse Dodge, and Kyle Lo. 2024. Dolma: An Open Corpus of Three Trillion Tokens for Language Model Pretraining Research. *arXiv preprint* (2024). doi:10.48550/arXiv.2402.00159

[63] The LlaVA Team. 2023. *LLaVA Data Documentation*. https://github.com/haotian-liu/LLaVA/blob/main/docs/Data.md

[64] The LlaVA Team. 2024. *TinyLLaVA: Model Zoo*. https://github.com/TinyLLaVA/TinyLLaVA_Factory?tab=readme-ov-file#model-zoo

[65] The Mosaic ML Team. 2022. *streaming: Fast, accurate streaming of training data from cloud storage*. https://github.com/mosaicml/streaming/

[66] The TensorFlow Team. 2025. *Tensorflow: Determinism*. https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_op_determinism

[67] The TinyLLaVA Factory Team. 2024. *TinyLLaVA Factory: Prepare Datasets*. https://tinyllava-factory.readthedocs.io/en/latest/Prepare%20Datasets.html

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[69] Zige Wang, Wanjun Zhong, Yufei Wang, Qi Zhu, Fei Mi, Baojun Wang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Data Management For Large Language Models: A Survey. *arXiv preprint* (2023). doi:10.48550/ARXIV.2312.01700

[70] Maurice Weber, Daniel Y Fu, Quentin Gregory Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, Ben Athiwaratkun, Rahul Chalamala, Kezhen Chen, Max Ryabinin, Tri Dao, Percy Liang, Christopher Re, Irina Rish, and Ce Zhang. 2024. RedPajama: an Open Dataset for Training Large Language Models. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*.

[71] Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy S Liang, Quoc V Le, Tengyu Ma, and Adams Wei Yu. 2024. Doremi: Optimizing data mixtures speeds up language model pretraining. *Advances in Neural Information Processing Systems* 36 (2024).

[72] Canwen Xu, Corby Rosset, Ethan C. Chau, Luciano Del Corro, Shweti Mahajan, Julian McAuley, Jennifer Neville, Ahmed Hassan Awadallah, and Nikhil Rao. 2024. Automatic Pair Construction for Contrastive Post-training. *arXiv preprint* (2024). 10.48550/arXiv.2310.02263

[73] Haoran Xu, Young Jin Kim, Amr Sharaf, and Hany Hassan Awadalla. 2024. A Paradigm Shift in Machine Translation: Boosting Translation Performance of Large Language Models. In *Proceedings of the ML Evaluation Standards Workshop at ICLR*. doi:10.48550/arXiv.2309.1167

[74] Juncheol Ye, Seungkook Lee, Hwijoon Lim, Jihyuk Lee, Uitaek Hong, Youngjin Kwon, and Dongsu Han. 2025. SAND: A New Programming Abstraction for Video-based Deep Learning. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. doi:10.1145/3731569.3764847

[75] Jiasheng Ye, Peiju Liu, Tianxiang Sun, Yunhua Zhou, Jun Zhan, and Xipeng Qiu. 2024. Data Mixing Laws: Optimizing Data Mixtures by Predicting Language Modeling Performance. *arXiv preprint* (2024). doi:10.48550/arXiv.2403.16952

[76] Xiang Yue, Yuansheng Ni, Tianyu Zheng, Kai Zhang, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, Cong Wei, Botao Yu, Ruibin Yuan, Renliang Sun, Ming Yin, Boyuan Zheng, Zhenzhu Yang, Yibo Liu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2024. MMMU: A Massive Multi-Discipline Multimodal Understanding and Reasoning Benchmark for Expert AGI. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr52733.2024.00913

[77] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. doi:10.1145/2934664

[78] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a Machine Really Finish Your Sentence?. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. doi:10.18653/v1/p19-1472

[79] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. *arXiv preprint* (2024). doi:10.48550/arXiv.2401.02385

[80] Juntao Zhao, Qi Lu, Wei Jia, Borui Wan, Lei Zuo, Junda Feng, Jianyu Jiang, Yangrui Chen, Shuaishuai Cao, Jialing He, Kaihua Jiang, Yuanzhe Hu, Shibiao Nong, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. 2025. OVERLORD: Ultimate Scaling of DataLoader for Multi-Source Large Foundation Model Training. *arXiv preprint* (2025). doi:10.48550/arXiv.2504.09844

[81] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. doi:10.1145/3470496.3533044

[82] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proceedings of the VLDB Endowment* 16, 12 (2023). doi:10.14778/3611540.3611569

[83] Donglin Zhuang, Xingyao Zhang, Shuaiwen Song, and Sara Hooker. 2022. Randomness in Neural Network Training: Characterizing the Impact of Tooling. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*.

Received July 2025; revised October 2025; accepted November 2025