Cachew: Machine Learning Input Data Processing as a Service

Dan Graur
ETH Zurich

Damien Aymon
ETH Zurich

Dan Kluser
ETH Zurich

Tanguy Albrici
ETH Zurich

Chandramohan A. Thekkath
Google

Ana Klimovic
ETH Zurich

Abstract

Processing input data plays a vital role in ML training, impacting accuracy, throughput, and cost. The input pipeline, which is responsible for feeding data-hungry GPUs/TPUs with training examples, is a common bottleneck. Alleviating data stalls is critical yet challenging for users. While today's frameworks provide mechanisms to maximize input pipeline throughput (e.g., distributing data processing on remote CPU workers and/or reusing cached data transformations), leveraging these mechanisms to jointly optimize training time and cost is non-trivial. Users face two key challenges. First, ML schedulers focus on GPU/TPU resources, leaving users on their own to optimize multi-dimensional resource allocations for data processing. Second, input pipelines often consume excessive compute power to repeatedly transform the same data. Deciding which source or transformed data to cache is non-trivial: large datasets are expensive to store, the compute time saved by caching is not always the bottleneck for end-to-end training, and transformations may not be deterministic, hence reusing transformed data can impact accuracy.

We propose Cachew, a fully-managed service for ML data processing. Cachew dynamically scales distributed resources for data processing to avoid stalls in training jobs. The service also automatically applies caching when and where it is performance/cost-effective to reuse preprocessed data within and across jobs. Our key contributions are autoscaling and autocaching policies, which leverage domain-specific metrics collected at data workers and training clients (rather than generic resource utilization metrics) to minimize training time and cost. Compared to scaling workers with Kubernetes, Cachew's policies reduce training time by up to 4.1× and training cost by 1.1× to 3.8×.

1 Introduction

Input data processing is an essential part of machine learning (ML) training. Transformations applied to input data before it is fed to a model for training – such as extracting features, sampling data from imbalanced classes, and randomly augmenting data – are key to achieving high accuracy [19, 57, 63]. Furthermore, the speed at which the input pipeline can ingest data from storage, apply transformations on-the-fly, and load transformed data to training nodes greatly impacts the time to accuracy and the overall cost of model training.

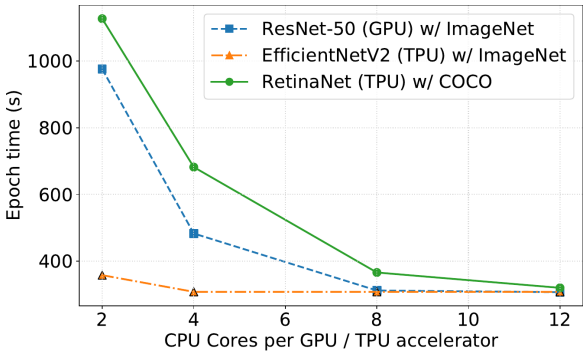


Figure 1: Training jobs benefit differently when given more CPU resources for input data processing per accelerator core.

While GPUs and TPUs used for training computations continue to provide more FLOPS, CPUs – which are responsible for input data processing – are not keeping up. Hence, the input pipeline is a common bottleneck in ML training [38].

Removing bottlenecks in the input pipeline can improve end-to-end training time by over an order of magnitude and greatly reduce costs [47, 48]. However, optimizing ML data processing is non-trivial. Users face several key challenges.

First, allocating the right amount of CPU, memory, and storage for input data processing, to optimize training time and cost, is difficult. Users should allocate *just enough* resources for the input pipeline to produce batches of data at the throughput that the model can ingest data, which depends on the model's computational intensity and the hardware (# of GPUs) allocated for training. As shown in Figure 1, each model requires a different ratio of CPUs for data processing and GPUs or TPUs for training. Hence, although ML frameworks traditionally couple input data processing and training such that the two stages execute on the same node, it is becoming increasingly common to disaggregate data processing, with systems like `tf.data` service [25] and Meta's Data PrePreprocessing (DPP) Service [69]. Disaggregation enables customizing resource allocations per job. However, today's ML resource management systems focus on GPU allocations [28, 44, 65], leaving users on their own to decide input

pipeline resource allocations that maximize performance and minimize cost of training jobs. This is notoriously challenging in the multi-dimensional resource setting of ML training [64].

Another major challenge is the significant amount of compute and memory resources required for input data processing. For example, when training deep recommender models with petabytes of data, data ingestion can consume more power than model training itself [69]. A promising approach to reduce data processing compute requirements is to memoize the outputs of commonly executed data pipelines, since ML training often involves redundant data accesses and transformations, both within and across jobs [34, 47, 48, 66]. Within a job, it is common to iterate multiple times (i.e., epochs) over a dataset. Across jobs, ML engineers typically experiment with variations of models (e.g., hyperparameter tuning and model search) while re-executing the same data pipeline.

ML data processing frameworks provide mechanisms for reusing memoized data transformations, such as `tf.data`'s `cache` and `snapshot` operators [22, 61, 62]. However, determining which (transformed) datasets are optimal to cache in fast storage is non-trivial. Transformed datasets are costly to store and slow to read if they are significantly larger in volume than source data, which can occur after decompression and data augmentations. Figure 2 shows that reusing memoized results stored on local SSDs of a training node does not always improve epoch time, since local SSD bandwidth may saturate when reading the larger transformed dataset. Even caching source data on local SSDs does not always improve epoch time compared to reading from cloud data lakes (e.g., we use GCS [27]) since model training may be the bottleneck. If an input pipeline applies random transformations to data each epoch, the caching decision is further complicated as reusing the transformed dataset from a single epoch can negatively impact model training dynamics [16, 40].

In summary, ML data processing frameworks provide useful *mechanisms* to alleviate data bottlenecks, such as distributing data processing on remote CPU workers and reusing cached data transformations. However, today's systems lack *policies* that efficiently leverage these mechanisms to optimize the overall performance and cost of ML training.

We propose Cachew, a fully-managed service for ML input data processing. Inspired by the serverless computing paradigm, which relieves developers from the burden of managing virtual machines in the cloud [13, 55], Cachew relieves ML users from the burden of managing compute, memory, and storage infrastructure for ML input data processing. Cachew consists of a centralized dispatcher, distributed input data workers, and a disaggregated storage cluster that stores cached datasets. We build Cachew on top of the `tf.data` framework [48], extending its distributed service [25] to support multi-tenancy, autoscaling, and autocaching. Cachew is open source[1] and compatible with the existing `tf.data` API.

[1] Cachew is available at: https://github.com/eth-easl/cachew

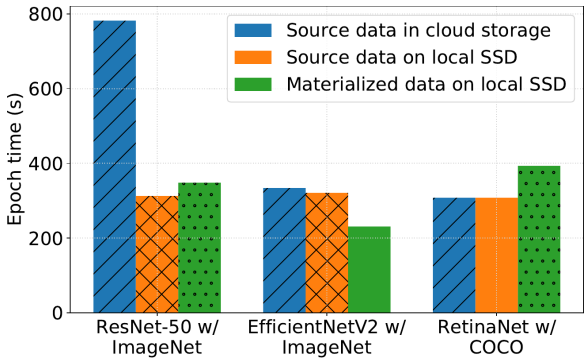


Figure 2: Caching source data or materializing data in local storage does not always improve training throughput.

Our key contributions are the design and implementation of autoscaling and autocaching policies for ML input data processing. We show that traditional resource autoscaling approaches (e.g., in Kubernetes [1]), which are based on CPU and memory utilization, are not sufficient to optimize training time and cost. Instead, we base our policies on application-specific metrics collected at input data workers and clients while a job is running. The Cachew dispatcher leverages the stateless nature of ML input data processing to adjust the number of workers per job during runtime. By monitoring batch time reported by clients, the dispatcher dynamically finds the minimum number of workers (i.e., minimum cost) that minimizes batch time (i.e., maximizes performance). By monitoring per-worker throughput and the volume of data produced by the input pipeline, the dispatcher also decides on whether reading data from Cachew's cluster cache is likely to improve performance compared to reading and transforming data from cloud data lakes on-the-fly. Cachew extends the `tf.data` API with an `autocache` operator, which allows users to specify up to which point in their input pipeline it is acceptable to cache and reuse data from a training dynamics perspective (e.g., before any random transformations).

We evaluate Cachew with microbenchmarks and three popular ML models and data pipelines from the TensorFlow Model Garden. We show that while the Kubernetes autoscaler under-provisons or over-provisions input data workers, Cachew is able to identify the optimal number of data workers to allocate for each job as well as the optimal caching strategy at each `autocache` operator location in the input pipeline to minimize training time and cost. We show that compared to scaling workers with Kubernetes, Cachew's policies reduce training cost by 1.1× to 3.8×.

2 ML Input Data Processing

We summarize the key characteristics of ML input data pipelines (§ 2.1) and discuss why it is increasingly common to disaggregate data processing from model training (§ 2.2). § 2.3 provides an overview of existing mechanisms for fast, efficient input data pipeline execution.

2.1 ML Input Data Pipeline Characteristics

Reading input data from storage: The first step in ML input data processing is reading source data. Deep learning input pipelines typically read datasets that range from gigabytes to petabytes in size, stored in low-cost distributed storage systems, such as cloud data lakes [48,68,69]. To avoid I/O bottlenecks during training, it is common to cache input datasets in more expensive, higher bandwidth storage systems [39,47].

Transforming data: Before raw input data can be consumed by a model, it must be preprocessed into elements that the model can learn from. Common transformations include decompressing data, parsing file formats, extracting features, and batching elements. It is also common to add randomness to input data (e.g., randomly sampling, augmenting, and shuffling elements) to improve model generalization [12]. Random data augmentations are critical for achieving state-of-the-art accuracy in image classification [14,15,18,21,58], object detection [19,67], and speech recognition [49,50].

Data transformations are generally executed on CPUs rather than specialized hardware to better support user-defined functions [48]. While some transformations can be applied in offline batch processing jobs [6], many transformations are applied *on-the-fly* during training for greater flexibility. For instance, it is common to experiment with feature extraction, tune the batch size for a given GPU configuration, or randomly augment data using different seeds across epochs.

While some transformations (e.g., decompression and augmentation) expand the volume of data read from storage, other transformations (e.g., filtering and sampling) decrease the volume of data fed to a model. A study of ML input pipelines at Google found that the ratio of data fed to a model versus the data read from storage varies widely across jobs. For 75% of jobs, data transformations reduced data volume [48].

Loading data to training nodes: The final step is to load data to GPUs/TPUs for model training. To avoid data stalls, the input pipeline must produce data at a throughput greater than or equal to the rate at which the model can consume data. The model's data ingestion rate depends on the algorithmic intensity of the training computations and the hardware FLOPS. Feeding data-hungry accelerators requires high software parallelism and pipelining for data processing [38,48].

Re-executing input pipelines: In large-scale production and research deployments, ML input data pipelines are commonly re-executed. Within a job, each training epoch reads and transforms the same input dataset. Across jobs, common ML training workflows, such as neural architecture search and hyperparameter tuning, involve feeding the same preprocessed data to different variations of a model [34,70]. At Google, the top 10% most commonly executed input pipelines accounted for 77% of all input pipeline executions and 72% of CPU cycles used for ML input data processing [48]. Others have also observed a sizeable opportunity to reuse data processing within and across ML jobs [47,66].

2.2 Why disaggregate input data processing?

ML input data processing and training consist of fundamentally different types of computation (user-defined data transformations vs. gradient updates) that primarily use different resources (CPUs vs. GPUs/TPUs). Yet ML frameworks have typically coupled these two stages, such that they run on the same nodes. Tight coupling has two major drawbacks. First, CPU/memory-intensive input pipelines can easily saturate host resources and limit training throughput. Zhao et al. [69] showed that loading data over the network from distributed storage – even without performing data transformations on the CPU – consumes significant CPU cycles and memory bandwidth in production deployments, leaving scarce resources available for transforming data on training nodes. Second, the ratio of resources required for input data processing vs. model training varies across jobs (as seen in Figure 1, where we varied the CPU cores available using the Linux `taskset` command). However, cloud providers typically limit the CPU cores and memory capacity that can be provisioned per accelerator on a virtual machine [9,26]. The fixed ratio of CPU cores and memory attached to accelerators often leads to imbalanced usage (i.e., either idle accelerators or idle CPUs).

To improve resource utilization and avoid input data stalls, it is increasingly common to disaggregate data processing from model training [25,69]. Disaggregation is a well-known approach for improving resource allocation flexibility [23,36]. This flexibility can save cost, since users can distribute input data processing across as many or as few CPU worker nodes as needed to avoid data stalls, without provisioning additional expensive GPUs/TPUs.

2.3 Existing Mechanisms

We describe key mechanisms in existing frameworks for efficient input data processing. Users can define and execute ML input data pipelines with a variety of data loading frameworks, such as `tf.data` [48], PyTorch DataLoader [17], NVIDIA DALI [29], and CoorDL [47]. We highlight the `tf.data` framework, as its combination of state-of-the-art mechanisms serve as a foundation for our work. `tf.data`'s programming model allows users to build input pipelines by composing and customizing operators. The framework runtime executes input pipelines as dataflow graphs, applying static and dynamic optimizations to improve performance.

Disaggregation: `tf.data` service supports executing input pipelines in a distributed manner [11]. The service consists of a centralized dispatcher and a number of remote input data workers. Clients (i.e., training nodes) register input pipelines defined in the `tf.data` API with the dispatcher, which shards data processing across all workers in the service. Clients fetch data directly from workers. The Data PreProcessing Service at Meta is another example of a framework that disaggregates input data processing [69]. In both frameworks, users are

responsible for managing the number of input data workers and deciding per-job resource allocations.

Dataset Caching: The `tf.data snapshot` operator allows users to cache the output of their input pipeline to disk, and materialize the transformed data on a subsequent training run [22]. This trades off I/O capacity and bandwidth to free up CPU resources. Inserting the `snapshot` operator at the appropriate point in an input pipeline to optimize overall training time and cost remains the user's responsibility. The current `snapshot` implementation does not coordinate between asynchronous reads and writes from multiple nodes, making it incompatible with `tf.data` service (hence, we implement our own `put` and `get` operators, described in §5.1).

CoorDL [47] and OneAccess [34] reuse input pipeline outputs when jobs are scheduled in a coordinated manner (e.g., hyperparameter tuning). However, these frameworks do not reuse data transformations across arbitrary ML jobs that can be submitted asynchronously to a service over time. Revamper [40] allows users to partially reuse the outputs of random transformations in input pipelines while minimizing the impact on training dynamics. Quiver [39] implements distributed caching for DNN training, but it is designed exclusively for managing source data rather than transformed datasets.

Autotuning: `tf.data`'s runtime and Plumber [38], a tool for diagnosing input data bottlenecks, can dynamically tune software parallelism and memory buffer sizes to maximize performance on a given training node [48]. However, this tuning does not scale resources beyond a single node.

GPU Offloading: NVIDIA DALI supports offloading certain input data processing, such as image data augmentations, to GPUs [29]. This is a viable option to alleviate CPU bottlenecks but may lead to GPU resource contention among input data transformation tasks and model training tasks. Users therefore need to decide which input data transformations should run on CPUs vs. GPUs.

3 ML Input Data Service Challenges

While many useful mechanisms for ML input data processing exist, it remains challenging for users to leverage these mechanisms to minimize end-to-end ML training time and cost. We focus on two key challenges: scaling resources for data processing (§ 3.1) and saving compute resources by selectively caching (transformed) datasets (§ 3.2).

3.1 Autoscaling Challenges

Selecting the right amount of compute, memory, and storage resources to provision for ML input data processing is critical yet challenging for ML users. Under-provisioning resources for data processing leads to data stalls, which leave expensive hardware accelerators idle, increasing end-to-end training time and cost. Over-provisioning resources for data processing leads to extra costs without improving end-to-end performance. The optimal resource allocation for data processing depends on the compute intensity of data transformations in the input pipeline, the volume of data that must be read for each training batch, and the rate at which the pipeline must produce data to match model ingestion throughput.

Determining the right resource allocation for ML input data processing is non-trivial as each model and input data pipeline combination have unique requirements [69]. For example, in Figure 1, we vary the amount of CPU cores allocated for input data processing to show how many cores are needed to meet model training throughput requirements in various jobs. To process the COCO dataset [43] and train the RetinaNet [42] model, 4 CPU cores per TPU accelerator core are sufficient, whereas to process the ImageNet [21] dataset and train the EfficientNetv2 [59] model, the user should provision 12 CPU cores per TPU accelerator core to avoid input data stalls during training. We also observe that scaling memory capacity and bandwidth greatly impacts performance. Furthermore, we find that training throughput does not scale linearly with CPU and memory resources, making it difficult to model how resource allocation affects performance.

3.2 Autocaching Challenges

Deciding which input data transformations to materialize and reuse versus which transformations to execute online during training is complex. At Google, where `tf.data` is heavily used in research and production ML training jobs, only 19% of jobs use any kind of caching operator [48]. Meanwhile, the same study found that many jobs would benefit from caching.

Making caching decisions requires users to reason about the cost of storing preprocessed data to save CPU cycles. This trade-off depends on the compute intensity of the input pipeline, the size of the materialized dataset, and the relative cost of CPU and storage resources. Transformed datasets may be slow to read and costly to store if they are significantly larger in volume than source data, which can occur with decompression and data augmentations. Caching does not always improve epoch time, in particular if reading source data from cloud storage and transforming it on-the-fly is sufficiently fast to saturate model ingestion throughput. Another challenge is that caching and reusing the results of transformations which randomly permute data from one epoch will remove this randomness across epochs. Reusing this transformed data within a job can significantly impact training dynamics. Prior work has shown that reusing random augmentations across epochs in a job is plausible, but must be done sparingly to avoid degrading model accuracy [16, 40]. For example, Revamper proposes caching partially augmented data elements and mixing them with freshly-computed, fully-augmented elements [40].

transformed data can be larger

An over-arching challenge is to jointly optimize autocaching and autoscaling. Regardless of whether input data

workers are reading and transforming source data on-the-fly from data lakes or reading transformed data from a cache, we need to determine the right number of input data workers and storage bandwidth to provision to maximize training throughput while keeping costs low.

4 Cachew Design

We introduce Cachew, a multi-tenant service for ML input data processing. To minimize end-to-end training time and cost, Cachew jointly optimizes: 1) elastic, distributed resource allocation for input data processing and 2) materialization of data processing computations within and across jobs. Cachew can be operated by an organization with multiple users that asynchronously submit ML training jobs or by a public cloud provider. With minimal extensions to the `tf.data` user API, Cachew transparently manages resource allocation for data processing, data caching, and network communication between Cachew clients and workers.

4.1 Service Architecture

Cachew consists of a centralized dispatcher, a dynamic number of input data workers, and a disaggregated storage cluster for data caching, as shown in Figure 3.

Users register training nodes (i.e., clients) of ML training jobs with the Cachew dispatcher. Clients provide a dataflow graph representation of their input pipeline and a path to the input data (§ 4.2). We assume input data resides in durable, low-cost cloud storage, i.e., data lakes [68] such as S3 [56].

Input data workers are stateless components responsible for producing batches of preprocessed data for clients. The dispatcher dynamically adjusts the number of input data workers for each job and divides each job's input dataset (e.g., a list of filenames) into independent partitions, called *splits*. Workers pull new splits (e.g., indexes of the file list) from the dispatcher when they are done processing previous splits. Workers may read splits that correspond to source data which they must transform on-the-fly by executing the job's input pipeline dataflow graph. Alternatively, splits may correspond to files that contain already transformed (or partially transformed) data in Cachew's cache from previous executions of the input pipeline.

The dispatcher maintains metadata about input pipeline executions across jobs to make worker allocation and data caching decisions. The scaling and caching policies (described in § 4.3) and § 4.4) rely on the metrics listed in Table 1, which the dispatcher aggregates in its *metrics metadata store*, indexed by job ids, input pipeline hashes and job names. Since there may be multiple workers and clients per job, metrics are averaged across clients and workers of the same job. The dispatcher also tracks which source and transformed datasets are cached. The *cache metadata store* maintains the location of cached datasets in Cachew's cluster cache and is indexed

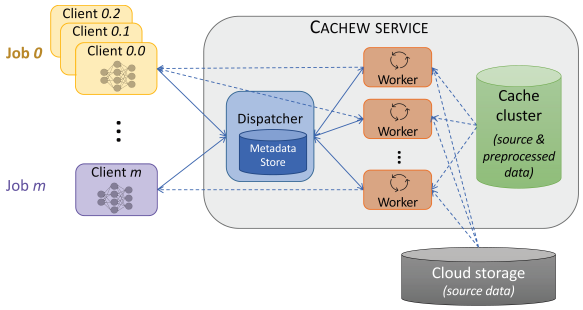


Figure 3: Cachew system architecture. Solid lines depict control logic and metadata communication. Dotted lines show the flow of training data. Communication occurs via RPCs.

```
1  dataset = tf.data.TFRecordDataset(["file1", ...])
2  dataset = dataset.map(parse).filter(filter_func)
3                   .autocache()
4                   .map(rand_augment)
5                   .shuffle().batch()
6  dataset = dataset.apply(distribute(dispatcherIP))
7  for element in dataset:
8    train_step(element)
```

Figure 4: User API to distribute `tf.data` input pipeline execution with Cachew. Users insert `autocache` to hint which data is acceptable to cache/memoize and reuse within a job.

by various hashes of the input pipeline dataflow graph, which we call fingerprints.

Clients fetch data from the workers that are assigned to them by the dispatcher. Clients and workers periodically send heartbeats to the dispatcher (by default every five seconds) to maintain membership in the service and provide metrics.

Cachew's cache cluster consists of high-bandwidth NVMe SSD storage nodes, which are disaggregated from input data workers. Hence, Cachew can scale storage independently, based on data caching capacity and bandwidth requirements. In addition to caching transformed datasets of frequently executed input pipelines, Cachew can also cache source datasets, to avoid I/O bottlenecks from cloud data lakes during training.

4.2 Cachew API

Cachew leverages the existing `tf.data` API for defining ML input data pipelines [48]. Users define a pipeline by chaining dataflow operators that can be parameterized with user-defined functions (UDFs). Figure 4 shows an example `tf.data` pipeline that reads input data from files, applies `map` and `filter` operators with UDFs for parsing, filtering, and randomly augmenting data, then shuffles and batches data.

Applying `distribute` in line 6 serializes the dataflow graph and sends it to the service dispatcher to register the job. If there are multiple training nodes in a job (i.e., for dis-

Source	Name	Description
Client	batch_time	Time taken to get and process the last 100 batches
	result_queue_size	Avg. number of batches located in the prefetch buffer over the last 100 batches
Worker	active_time	Avg. time per element spent in computation in the subtree rooted in the node
	bytes_produced	Total number of bytes produced by the node so far
	num_elements	Total number of elements produced by the node so far

Table 1: The set of metrics that are submitted by the workers and clients to the dispatcher via heartbeats.

tributed ML training), each node registers as a separate client with the dispatcher and specifies an additional `job_name` parameter that is common across all clients of the same job. The communication between clients and workers is abstracted away from the python API. As shown in line 7, clients simply iterate over the `dataset` to access a sequence of elements, as if executing the input pipeline locally on the client.

Cachew introduces a new operator, `autocache`, which allows users to specify point(s) in an input pipeline where model training dynamics safely permit memoizing and reusing data within a job. To avoid any impact on training accuracy, users should apply `autocache` before any random data transformations in their pipeline (e.g., see line 3 of Figure 4). Users may apply `autocache` in multiple locations in a pipeline. Cachew will not always apply caching at an `autocache` operator; § 4.4 describes Cachew's decision strategy. When `autocache` is placed after a read operator (e.g., line 1), Cachew decides whether caching source data in fast cluster storage improves performance compared to reading source data from a low-cost data lake. In §6, we evaluate Cachew with up to two `autocache` operators per pipeline.

4.3 Autoscaling Policy

We describe how Cachew leverages the per-job client metrics described in Table 1 to make worker scaling decisions.

Allocating workers for new jobs: The dispatcher starts by executing each new job with a single worker. After the client reports metrics from a configurable number of training batches, the dispatcher allocates a second worker for the job and monitors the change in `batch_time`. We observe that averaging metrics over 100 batches generally provides a satisfactory level of noise smoothing. If `batch_time` decreases by more than a threshold (which we empirically set to 3%) with the second worker, the dispatcher adds an additional worker. The dispatcher continues adding workers until a new worker improves `batch_time` by less than 3%, in which case the scaling decision converges. The epoch time (and hence batch time) plateaus when the workers can provide data at sufficient throughput to saturate the model ingestion rate (e.g., the red dotted line in Figure 6). The true plateau occurs when the addition of a new worker leads to 0% change in `batch_time`. However, we choose a slightly higher threshold due to the noisy nature of metrics gathered at runtime. This makes Cachew's autoscaling policy more stable. We observed that a 0% threshold would lead to unstable scaling decisions, as the slightest noise could trigger the addition of superfluous workers or the removal of essential workers from the cluster. In order to reduce noise, it is possible to gather the metrics over more batches, however this slows down the scaling process. §6.2 shows a sensitivity study.

Re-scaling over time: Since metrics can be noisy, Cachew periodically revisits the scaling decision each 10 new metrics received from the client (i.e., every 1000 batches). Cachew adds a worker if the current `batch_time` is significantly higher than the value recorded at scaling convergence. To detect if we are on the batch time plateau and can afford to scale down, our intuition is that although batch time will be the same, the result queue will build up if workers are able to provide data faster than the model can ingest it. Hence, Cachew removes a worker if the current `result_queue_size` is significantly (e.g., $> 40\%$) higher than the size recorded at convergence. Cachew continues removing workers as long as the increase in `batch_time` is below the threshold. Clients temporarily pause metric collection (e.g. 150 batches) when they are notified that a worker has been added or removed for their job. When the dispatcher de-allocates a worker from a job, the worker processes any remaining splits, which clients consume before the worker is removed. Hence, the model sees all data in an epoch, regardless of scaling events.

We observed that in some cases, when the input dataset consists of few files (e.g., the COCO dataset), the scaling policy may not converge within an epoch because workers may prefetch all dataset splits, leaving no splits for newly added workers to process until the next epoch. Cachew detects these scenarios and inserts an artificial epoch, which allows workers to fetch data from the next epoch while other workers process data from the current epoch. We do not introduce artificial epochs if workers are writing to cache.

4.4 Autocaching Policy

When a new job is sent to Cachew, a hash of the entire input pipeline is generated, which is then used to check if the given pipeline has ever had some of its data cached. If yes, the dispatcher extracts a hash for each `autocache` op. This is done by traversing and hashing the input pipeline from its source nodes up until the `autocache` node. Cachew checks its

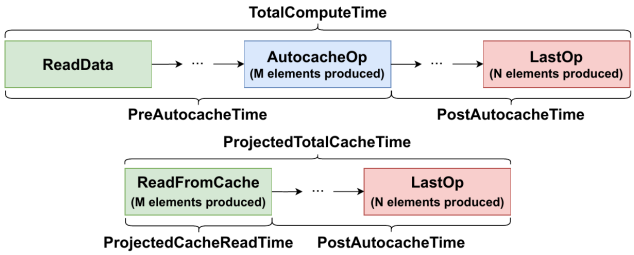


Figure 5: Cachew autocaching policy calculation.

cache store against these hashes to find potential hits. Cachew will choose to introduce caching at the `autocache` location with the highest throughput. It should be noted that in such cases, compute is not considered, as caching only occurs if it was deemed better in terms of throughput than compute. If the given pipeline has not got any of its data in cache, the job enters profiling mode. This stage is equivalent to full input pipeline computation, with the exception that scaling is blocked to not skew any metrics relevant to the autocaching decision. The relevant metrics are presented in the Worker section of Table 1. Once the input pipeline has produced a sufficient amount of batches (we observe 300 batches to be enough) Cachew's dispatcher produces the autocaching decision.

To carry out the autocaching decision, for each `autocache` operator, the dispatcher estimates the time it takes the pipeline to produce N elements if caching were to be introduced at the `autocache` location. The dispatcher compares these N-element-times with the compute mode N-element-time (i.e., how long it takes to produce N elements if no caching is applied to the pipeline) and selects the option with the minimum time. If caching is chosen, Cachew introduces caching at the relevant `autocache` op location. Figure 5 shows the main values inferred by the dispatcher for the autocaching decision. It should be noted that due to operations such as `batch`, `filter`, or `repeat`, an `autocache` op sees M elements being produced, where $M \neq N$ can be true.

Let the `LastOp`'s `active_time` be $\texttt{active\_time}_L$, an `autocache`'s `active_time` be $\texttt{active\_time}_A$ and the bytes per element at an `autocache` op be b_A. Cachew employs a throughput model of GlusterFS, formally defined as $g_{GFS} : \mathbb{R}_{>0} \rightarrow \mathbb{R}_{>0}$, which can infer the read time of b bytes from cache. Cachew initially computes the `TotalComputeTime` = $N \times \texttt{active\_time}_L$. For each `autocache` op, Cachew computes the `PreAutocacheTime` = $M \times \texttt{active\_time}_A$, and then obtains the `PostAutocacheTime` = `TotalComputeTime` − `PreAutocacheTime`. Next, Cachew computes `ProjectedCacheReadTime` = $M \times g_{GFS}(b_A)$. Finally, the `ProjectedTotalCacheTime` = `ProjectedCacheReadTime` + `PostAutocacheTime` is computed. The `ProjectedTotalCacheTime` of each `autocache` op and the initial `TotalComputeTime` are compared, and the option with the lowest value is selected.

Once the decision is made, scaling is re-enabled. Scaling is triggered when the execution policy of the input pipeline changes (e.g. from putting to getting data from cache). When this happens, the worker count of a job is set to 1, and the autoscaling policy is applied by initially scaling up.

4.5 Multi-tenancy

Cachew supports multi-tenancy. Jobs submitted by different clients are accepted by Cachew, whose dispatcher applies the autoscale and autocache policies on each job. The jobs are self contained, and the autoscale and autocache decisions are independent of other jobs running in Cachew. To ensure such decisions are correct, metrics from different jobs with identical input pipelines are stored in the metadata store separately from one another using the job names. Moreover, to avoid performance interference, Cachew assigns each workers to at most one job at a time.

Our current prototype of Cachew assumes that tenants are mutually trusted and have permission to access each other's data. Hence, Cachew shares cached datasets across jobs from different tenants. In §7, we discuss how the implementation can be extended to implement data access control.

4.6 Fault tolerance

Dispatcher: The dispatcher stores metadata in memory for fast look-ups. To avoid being a single point of failure, the dispatcher can journal its state to a durable directory, such that no state is lost when the dispatcher is restarted. Journaling is also supported in the vanilla `tf.data` service [25].

Workers: Our implementation builds on top of existing single-node tf.data checkpointing mechanisms, extending them to work in the distributed setting. Workers communicate with the dispatcher via heartbeats. If a parametrizable number of heartbeats are missed (we set this parameter to 2), the dispatcher considers the worker failed and initiates a failover. The dispatcher reassigns pending tasks to a free worker in the pool. The new worker recovers any progress the failed worker had made on the pending tasks by reading the latest checkpoint committed in remote storage. The new worker recomputes batches between the latest checkpoint and the time of failure, to make the new worker's iterator state match the old worker's iterator state at the time of failure. The client includes an incrementing index in its request which the new worker uses as time of failure to fast forward to. However, the new worker does not transmit the recomputed batches to the client since repeating data elements in training epochs can harm model accuracy. For instance, Mohan et al. observed a double-digit drop in Top-1 accuracy when training a ResNet18 for 70 epochs on ImageNet-1k without exactly-once semantics [46]. In contrast, Cachew guarantees that clients see each batch of input data exactly once during

training, even in the face of failures. Furthermore, our fault-tolerance mechanisms make it viable to run Cachew workers on transient cloud resources (i.e., spot VMs) to reduce the cost of using remote workers for data processing.

Storage nodes: Cachew's distributed cache applies erasure coding with configurable redundancy. By default, we configure the storage layer to store data with sufficient redundancy to handle up to 25% of nodes in the storage cluster failing at a given time. If a storage node fails, the distributed file system uses parity blocks to recover data.

5 Implementation

We implement Cachew on top of the `tf.data` ML data processing framework [48], leveraging its familiar API and mechanisms for distributed data processing and dataflow graph rewriting. We also add a scalable cluster cache to the vanilla `tf.data` service and expose the `autocache` operator to users in the API. We extend the `tf.data` service dispatcher with metadata stores that manage client/worker metrics and locations of cached datasets to implement our scaling and caching policies. Our implementation consists of approximately 9000 lines of C++ code and 500 lines of Python on top of the open-source `tf.data` code base. Cachew is open-source.

We run the Cachew dispatcher and workers inside Docker containers and use Kubernetes to elastically scale the deployment. All communication between clients, workers, and the dispatcher is done over gRPC [24]. We use GlusterFS [53], deployed on high-bandwidth NVMe SSD storage nodes, as our distributed caching storage system. GlusterFS is highly scalable, offers sufficient throughput to saturate NVMe SSD bandwidth, and its consistent hashing data distribution policy supports dynamically adding and removing storage nodes. We configure GlusterFS with distributed dispersed volumes, which use erasure coding for fault-tolerance [54].

5.1 `Autocache` Mechanisms

Graph rewrites: When a user registers a new job, Cachew's dispatcher inspects the input pipeline's dataflow graph. Wherever the pipeline contains an `autocache` op, Cachew generates two versions of the pipeline, in which the `autocache` is replaced by a `put` op and a `get` op, respectively. The dispatcher transparently replaces `autocache` with graph rewrites using the TensorFlow Grappler [60] optimization framework. The dispatcher sends workers the correct version of the input pipeline graph, which depends on the execution mode selected by Cachew's autocaching policy for the job.

The `put` and `get` ops: Cachew introduces `put` and `get` ops which store and retrieve data to and from the cache. These ops build on underlying mechanisms in the `tf.data snapshot` op implementation, but are designed for multi worker scenarios, where several different worker nodes can concurrently write and/or read to the same cache location without conflicts. To support asynchronous behaviour, the `put` and `get` op implementations both leverage queues and multi-threading. For the `put` op, data to be written to cache is placed in a queue. Multiple threads greedily dequeue elements and write them to cache. Each thread writes in its own file, which is closed when the size exceeds 250MiB and a new file is opened. The `get` op functions in the opposite manner, where threads read from cache and place elements in the queue. When downstream operations require data from the `get` op, they dequeue elements. Each thread requests the file paths to read from the dispatcher. Operations before the `get` op are not executed since the output is read from cache.

Since the ops are asynchronous, reads and writes will be performed out of order. Hence, cache reads and writes behave as a sliding-window shuffle of size $w_s = s \times n$, where w_s is the window size, s is the number of elements per cache file and n is the number of readers or writers.

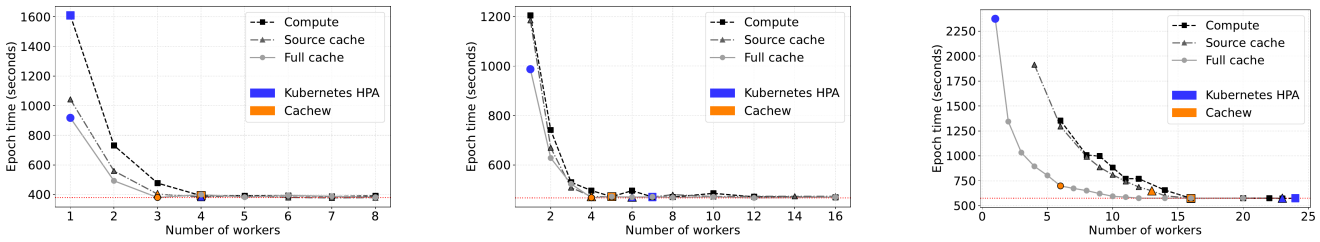
Dealing with limited cache capacity: Our current prototype assumes that Cachew's cache capacity is unbounded (i.e., there is always space in the cache for a `put` op to succeed). To operate Cachew with a limited cache size, we plan to extend the dispatcher implementation to periodically evict cached datasets that provide the least performance improvement across jobs, as in prior caching systems like Nectar [30].

6 Evaluation

6.1 Methodology

Workloads: We evaluate Cachew with three popular ML models and their corresponding input data pipelines in TensorFlow Model Garden [3]. ResNet-50 [31] is an image classification model whose input pipeline consists of parsing raw TFRecord files, converting images to float16 format, and applying a random crop and horizontal flip [4]. We use ImageNet [21] and observe that the data transformations increase the source dataset by 2.6×. RetinaNet [42] is an object detection model whose input pipeline consists of parsing TFRecords, converting images to float16, applying a random horizontal flip, and a series of computationally-intensive operations that create candidate anchors at five different scale levels [5]. We train RetinaNet on the COCO [43] dataset and the data transformations increase the data volume by 32.6×. Finally, SimCLRv2 [15] is a semi-supervised learning framework used for visual representation learning model. Given a randomly sampled mini-batch of images, each image is augmented twice with a random crop, color distortion, and Gaussian blur, creating two views of the same example. The model learns representations by maximizing agreement between differently augmented views of the same data example [2]. We use SimCLR for semi-supervised image classification on ImageNet. The data transformations increase the data volume by 10.7×.

Baselines: We compare Cachew's resource scaling policy with the Kubernetes Horizontal Pod Autoscaler (HPA)



(a) ResNet50 (b) RetinaNet (c) SimCLR

Figure 6: Scaling policy. Cachew selects the right number of workers to minimize epoch time and cost (orange markers). Kubernetes Horizontal Pod Autoscaler does not select the optimal number of workers (blue markers), since it only scales based on CPU usage and does not account for other potential input pipeline bottlenecks, e.g., memory and I/O bandwidth.

policy, which scales input data workers in the service based on a 80% CPU resource utilization target per node [1]. We measure the best-case epoch time for each model (i.e., model ingestion rate) by running an infinitely fast input pipeline that feeds synthetic data. Finally, we report the overhead of running `tf.data` pipelines on remote workers versus on training nodes.

Execution modes: We consider two different placements of the `autocache` operator in our input pipelines. We assume the user inserts `autocache` near the beginning of the input pipeline, immediately after a data read operator and before any data transformation operators. We call this placement *source cache mode* since it causes data workers to read source data from Cachew's cluster cache if the dispatcher decides to apply caching at this point in the pipeline. We also assume the user inserts `autocache` at the end of the input pipeline, after all data transformations. We refer to this placement as *full cache mode*, since input data workers will read fully transformed data from the Cachew cluster cache if the dispatcher decides to apply caching at this point in the pipeline. We compare the performance of the source cache and full cache execution modes with a *compute mode*, in which no data is reused in the input pipeline, i.e., Cachew reads source data from cloud storage and transforms data on-the-fly.

Metrics: We measure epoch time, i.e. the time it takes to train the model on a complete iteration of the dataset, for each model while varying the number of input data workers and execution modes. We also report total training time and cost for the compute and source cache execution modes. For cost, we consider the Cachew input data worker node costs, storage resources, and the cost of training nodes used for the duration of the job. We assume the cost of the dispatcher is amortized across multiple users and jobs. Note that in our workloads, we observed full caching decreases training accuracy due to the presence of random transformations in the input pipelines. In our evaluation, we focus on demonstrating that Cachew can select the right execution mode to maximize throughput, assuming the user has placed the `autocache` operator in a location that is acceptable for their use-case. Prior studies have explored the impact on training dynamics when reusing randomly transformed data across epochs [16, 40].

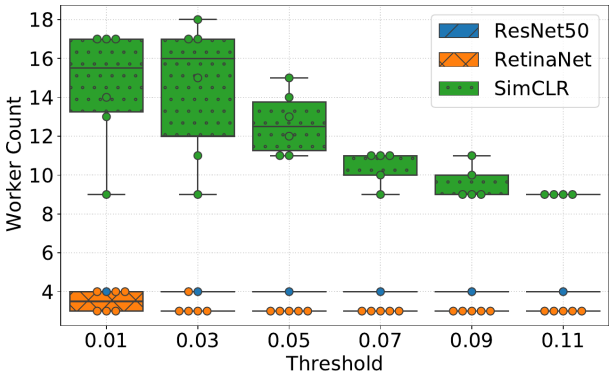


Figure 7: Cachew's first scaling decisions in compute mode relative to the value of the improvement threshold.

Cluster hardware setup: We run our experiments on Google Cloud. The dispatcher runs on a n2-standard-16 VM. We train ResNet-50 on n1-standard-32 VMs with four Tesla V100 GPUs. We train SimCLRv2 and RetinaNet models on v3-8 TPU VMs since the reference implementations are designed for training on TPUs. We use n2-standard-2 VMs with two 375GB NVMe SSDs for the GlusterFS storage cluster. The network bandwidth between the storage cluster, workers, and clients is at least 16 Gb/s.

6.2 Cachew Autoscaling

We sweep the number of input data workers for the compute, source cache, and full cache execution modes for each model. Figure 6 plots epoch time as a function of the number of data workers and shows the number of workers selected by Cachew's scaling policy in orange and Kubernetes's scaling policy in blue markers. The dotted red line shows the minimum epoch time achievable with an infinitely fast input pipeline. In all execution modes, Cachew finds the minimum (or near minimum) number of data workers to avoid data stalls

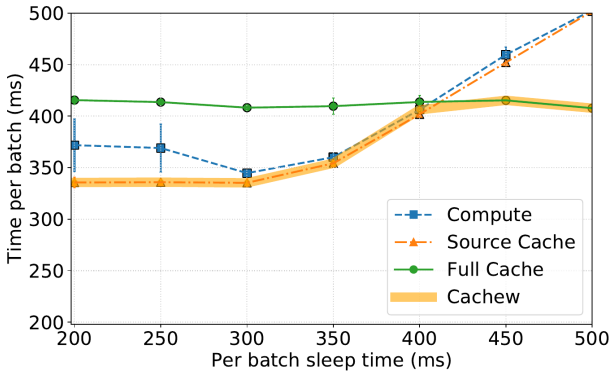


Figure 8: Cachew's autocaching policy selects the execution mode that minimizes batch time.

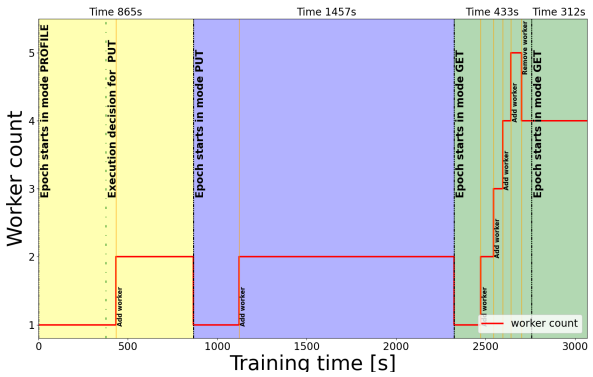


Figure 9: RetinaNet training timeline for the first 4 epochs. Cachew picks the right caching mode and number of workers.

in model training. In contrast, the Kubernetes Horizontal Pod Autoscaler noticeably under or over-provisions data workers. Kubernetes HPA performs poorly as it does not detect bottlenecks besides CPU and memory capacity, such as memory bandwidth bottlenecks and I/O bottlenecks that can limit input pipeline throughput. For SimCLRv2, the input pipeline is highly compute intensive, hence Kubernetes scales up to 24 data workers to maintain per-node CPU utilization at the 80% target. In contrast, Cachew's scaling policy checks the relative improvement in batch time and determines there are diminishing returns to scaling beyond 16 workers in compute mode and 13 workers in source cache mode.

Figure 7 shows Cachew's scaling decision sensitivity to the `batch_time` improvement threshold value, which we sweep from 1% to 11%. We include decisions that are suboptimal, which Cachew may later correct in the training process. Both ResNet50 and RetinaNet are robust to this threshold, as the addition of a worker yields clear epoch time benefits (see Figures 6a and 6b), thus the value of the threshold can be quite large. For SimCLR, the addition of a new worker does not always yield significant benefits to the epoch time (see Figure 6c). Consequently, for such a model, lower thresholds are more suitable. The downside of a lower threshold is that the autoscaling policy becomes more susceptible to noise in the metrics. This is visible in Figure 7, as the variance of the decision increases, and outlier decisions become more common. Cachew triggers rescaling in such cases, and eventually converges to the right decision. Gathering metrics over multiple batches can help alleviate noise.

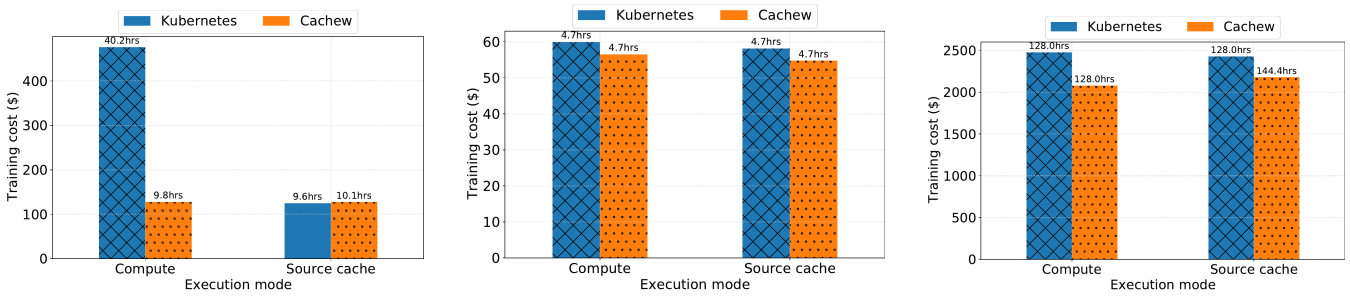
6.3 Cachew Autocaching

To evaluate Cachew's caching policy, we run the service with an input pipeline in which we carefully control and simulate compute intensity. The input pipeline consists of reading input data from storage (56GB), increasing the size of data elements by 2.5×, and sleeping for a controlled duration of time to simulate applying a time-consuming data processing operation. Figure 8 plots the time it takes to process a batch of elements as a function of the injected processing time, for the three different execution modes we consider. The experiment is designed such that the compute, source cache, and full cache execution modes are each optimal in a particular regime and verify that Cachew makes the optimal choice.

At low data processing intensity, reading data from the cloud data lake (GCS) is an I/O bottleneck and Cachew recognizes the service should store and read source data from the cluster cache. When sleep exceeds 350ms, Cachew recognizes that data processing intensity is high enough that GCS I/O is no longer the bottleneck. Reading and transforming data on-the-fly from GCS (i.e., compute mode) becomes optimal until 400ms. When source cache mode reaches a similar throughput as compute mode (as in the 350-400ms regime), Cachew prefers compute mode as it saves storage costs. When sleep time exceeds 400ms, storing and reading the transformed dataset from cache minimizes batch time compared to other modes. Cachew chooses full caching in this regime.

6.4 Autocaching & Autoscaling over Time

We demonstrate how Cachew jointly optimizes elastic scaling and caching to maximize epoch time and cost. Figure 9 plots the scaling and caching decisions that Cachew makes over time. As an example, we show the first four epochs of RetinaNet training, where we place two `autocache` ops: one after the reading ops, and one at the end of the input pipeline. The red curve shows the number of workers used for the job and orange vertical lines show when Cachew makes a scaling decision. Cachew starts by processing the pipeline in compute mode with a single worker (highlighted in yellow), first blocking scaling and profiling the worker. During the single-worker profiling phase, Cachew collects metrics to make a caching policy decision and decides on caching at the end of the input pipeline. This decision takes place at the time marked by the green vertical dashed line. The caching decision is applied at the end of the first epoch, when we enter the `put` epoch



(a) ResNet50 (b) RetinaNet (c) SimCLR

Figure 10: Total training cost (and training time) for Cachew vs. Kubernetes HPA worker scaling policy decisions.

(highlighted in blue) which writes the dataset to the cluster cache and takes longer as a result. Due to prefetching, splits are exhausted in the first two epochs before the autoscale completes. The other two epochs presented are in cache `get` mode (highlighted in green). In the third epoch, Cachew scales the number of workers up to 5, where no more improvements are observed. At this point Cachew removes the superfluous worker and converges to 4 workers. Cachew will continue to run the pipeline in `get` mode with 4 workers for the remaining epochs, and only adjust this number if input pipeline characteristics evolve, requiring less or more workers.

6.5 End-to-end performance and cost

6.5.1 Time to Accuracy and Training Cost

Figure 10 shows the total training cost (bars on y-axis) and total training time (annotations) for the input data worker configurations that Kubernetes and Cachew select in the compute and source cache execution modes. As we saw in Figure 6, Kubernetes under-provisions input data workers for ResNet-50 compute mode, while Cachew picks the optimal worker count, allowing it to reduce training time by 4.1× and cost by 3.8×. For SimCLR, Kubernetes over-provisions workers, slightly reducing training time but Cachew still saves overall cost by 12-20% by optimizing the number of workers based on its relative improvement threshold in the scaling policy.

6.5.2 Service Overhead

We compare the performance and resource overhead of running `tf.data` pipelines with the service versus locally on training nodes. The service requires additional CPU resources to achieve the same input pipeline throughput, due to the extra network hop (i.e., gRPCs) between data workers and training nodes. Since local `tf.data` workers fetch source data over the network and transform data on the training node, whereas Cachew clients fetch transformed data over the network, the service overhead also depends on the difference between the source and transformed dataset sizes. In our experiments, we find that the service should provision 30% to 50% more CPU

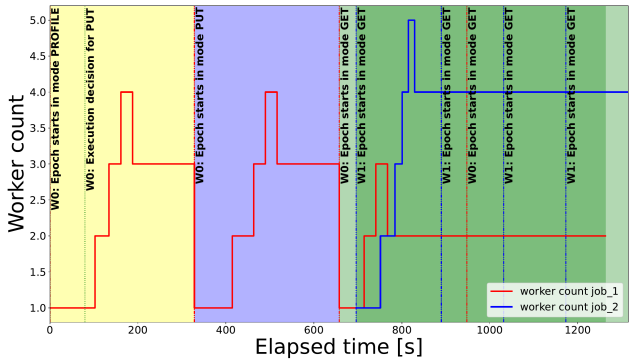


Figure 11: Multi-tenancy: Cachew detects cache hits across jobs while scaling workers for each job separately.

cores compared to the CPU cores we observe being heavily utilized when running the same input pipeline locally on training nodes. Note that this overhead is not an artifact of Cachew's scaling or caching policies, but rather a measurement of the overhead Cachew inherits from the distributed data processing mechanisms in `tf.data` service [25]. Data marshaling is known to consume significant cycles in datacenters [35]. Networking overheads can be reduced (e.g., by using RDMA), though this is not the focus of our work. The cost of using extra CPUs for data processing is justified as it allows keeping expensive GPUs/TPUs highly utilized and reducing end-to-end training time (see Figure 10).

6.6 Multi-tenancy

We show that Cachew can optimize input data processing across jobs. We run two jobs with the ResNet input data pipeline but different model ingestion rates (simulated with different sleep times in the dataset iteration loop on the clients). The second job has double the ingestion rate of the first. We again place `autocache` ops after reading the data and at the end of the input pipeline. Figure 11 shows how the batch processing time varies over time for each job as Cachew makes its scaling and execution mode decisions. The red

curve shows the first job's number of workers over time. The epoch boundaries are highlighted by the red dotted vertical lines. This job progresses through a compute epoch (yellow highlight) with profiling, a put epoch (blue highlight), where it caches the data of the `autocache` at the end of the input pipeline and a get stage (green highlight), which consists of two get epochs. The job ultimately converges to two workers.

We show that the dispatcher detects a cache hit after the first job has written to cache and selects the cache `get` execution mode for the second job in all of its four epochs. The epochs are separated by blue dotted vertical lines. The blue curve shows the number of workers for the second job over time. As this job has double the ingestion rate requirements of the first job, it converges to four workers.

7 Discussion

data access problem

Data access control: Our current prototype of Cachew assumes that clients have permission to access each other's data (e.g., tenants are all members of the same organization). The implementation can be extended with Access Control Lists (ACLs) to prevent unauthorized access to the data. Workers can check/set permissions before reading/writing datasets. If a job attempts to read from a dataset for which the client does not have read permission, the worker's request will fail and the client will get a permission error.

Model training dynamics: Though reusing the output of random data augmentations can negatively impact model accuracy, prior works have found that negative impacts can be mitigated by tuning the extent to which caching is applied in an input pipeline. Choi et al. showed that reusing data after random transformations has a small negative impact on the final accuracy trained models, reaching highly competitive out-of-sample error rates with fewer non-cached data instances than a model with no echoing [16]. Revamper [40] demonstrated that partially caching random transformations, while leaving some to be applied after reading from cache, has negligible accuracy penalties, as long as the downstream random transformations provide sufficient sample diversity. Hence, a good rule of thumb is to cache expensive random transformations, while applying highly diverse and inexpensive random transformations after the cache. For instance, in tasks such as image classification, inexpensive random transformations, such as random crop and flip, generally provide sufficient sample diversity [40].

Leveraging local resources on training nodes: Although our autoscaling policy has focused on leveraging remote CPU workers for data processing, `tf.data` service also supports running data processing on local workers, which execute on client training nodes. Adapting Cachew's autoscaling policy to leverage a mix of local and remote workers would ensure that the extra cost of remote workers is only incurred if the CPU/memory resources for data processing on client training nodes are not sufficient to avoid data stalls.

8 Related Work

§ 2.3 discussed existing mechanisms in input data frameworks, which serve as the foundation for Cachew's autoscaling and autocaching policies. We discuss other related work below.

Automated resource provisioning: Cluster management systems aim to automate resource allocation decisions, which are notoriously difficult for users [8, 20, 37, 45]. Generic cluster managers treat workloads as black boxes, making them unsuitable for jointly optimizing data caching and resource scaling decisions. Cachew is able to jointly optimize caching and scaling by using metrics that are tailored to the ML input data processing domain. Several resource management systems have been developed specifically for deep learning jobs [28, 44, 65]. However, they assume GPUs are the dominant resource for ML training, whereas recent work has shown that allocating resources for input data processing is equally important yet not well addressed by existing systems [64].

Caching vs. recomputing intermediate data: Caching data and memoizing data transformations is a common technique [10, 41, 52]. Trade-offs of caching vs. recomputing data arise in various contexts [7, 32, 33, 51]. We draw particular inspiration from Nectar [30], a datacenter-scale caching system that treats computations and their intermediate results as interchangeable. We address ML-specific challenges, where estimating the benefit of caching on training time is non-trivial since the model may be the bottleneck or the (transformed) dataset may be prohibitilely large to cache. We also jointly optimize caching and resource scaling.

9 Conclusion

We proposed Cachew, a system architecture, to enable input data processing *as a service* for machine learning. To avoid input data stalls while minimizing cost, Cachew dynamically scales distributed data processing resources to match the rate at which each job's training nodes can ingest data, while avoiding over-provisioning. Cachew leverages its centralized view of data processing pipelines across mutually trusted jobs to reduce the overall compute power required for data processing, by transparently reusing (transformed) datasets within and across jobs when performance and cost efficient.

Acknowledgements

We thank our anonymous reviewers and shepherd for their valuable comments. We acknowledge Jiří Šimša, Andrew Audibert, Gustavo Alonso, Petros Maniatis, Paul Barham, and Julia Bazińska for discussions that helped strengthen this work. We also thank Oto Mraz for his help with artifact evaluation. We are grateful for access to the Google TPU Research Cloud and generous support from a Google Research Award.

References

[1] Kubernetes Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2021.

[2] SimCLR reference code. https://github.com/google-research/simclr, 2021.

[3] TensorFlow Model Garden. https://github.com/tensorflow/models, 2021.

[4] TF Model Garden: ResNet50 reference code. https://github.com/tensorflow/models/tree/master/official/vision/image_classification/resnet, 2021.

[5] TF Model Garden: RetinaNet reference code. https://github.com/tensorflow/models/tree/master/official/legacy/detection, 2021.

[6] Data preprocessing for machine learning: options and recommendations. https://cloud.google.com/architecture/data-preprocessing-for-ml-with-tf-transform-pt1, 2022.

[7] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated selection of materialized views and indexes in sql databases. In *In Proceedings of VLDB '00*, pages 496–505, 2000.

[8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017.

[9] Amazon Web Services. AWS EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2021.

[10] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proc. of Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, 2012.

[11] Andrew Audibert and Rohan Jain. tf.data Service RFC. https://github.com/tensorflow/community/blob/master/rfcs/20200113-tf-data-service.md, 2019.

[12] Leon Bottou. Curiously Fast Convergence of some Stochastic Gradient Descent Algorithms. In *Proceedings of the Symposium on Learning and Data Science*, 2009.

[13] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, 2019.

[14] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In Hal Daumé III and Aarti Singh, editors, *Proc. of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1597–1607. PMLR, 13–18 Jul 2020.

[15] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey E Hinton. Big self-supervised models are strong semi-supervised learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 22243–22255. Curran Associates, Inc., 2020.

[16] Dami Choi, Alexandre Passos, Christopher J. Shallue, and George E. Dahl. Faster Neural Network Training with Data Echoing, 2019.

[17] Torch Contributors. PyTorch Docs: torch.utils.data. https://pytorch.org/docs/stable/data.html, 2021.

[18] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation strategies from data. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 113–123, 2019.

[19] Ekin Dogus Cubuk, Barret Zoph, Jon Shlens, and Quoc Le. Randaugment: Practical automated data augmentation with a reduced search space. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, pages 18613–18624, 2020.

[20] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, 2014.

[21] Jia Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of CVPR*, 2009.

[22] Frank Chen and Rohan Jain. tf.data Snapshot RFC. https://github.com/tensorflow/community/blob/master/rfcs/20200107-tf-data-snapshot.md, 2020.

[23] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, November 2016.

[24] Google. gRPC: a high performance, open source universal RPC framework. https://grpc.io/, 2021.

[25] Google. Module: tf.data.experimental.service. https://www.tensorflow.org/api_docs/python/tf/data/experimental/service, 2021.

[26] Google Cloud. Cloud TPU pricing. https://cloud.google.com/tpu/pricing, 2021.

[27] Google Cloud. Google Cloud Storage. https://cloud.google.com/storage, 2021.

[28] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, February 2019.

[29] Joaquin Anton Guirao, Krzysztof Łęcki, Janusz Lisiecki, Serge Panev, Michał Szołucha, Albert Wolant, and Michał Zientkiewicz. Fast AI Data Preprocessing with NVIDIA DALI. https://devblogs.nvidia.com/fast-ai-data-preprocessing-with-nvidia-dali, 2019.

[30] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proc. of Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of CVPR*, pages 770–778. IEEE Computer Society, 2016.

[32] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proc. of Programming Language Design and Implementation (PLDI'00)*, page 311–320, 2000.

[33] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. *Proc. VLDB Endow.*, 11(7):800–812, March 2018.

[34] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[35] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proc. of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[36] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proc. of European Conference on Computer Systems*, EuroSys '16, pages 29:1–29:15, 2016.

[37] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, July 2018. USENIX Association.

[38] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. In *Proc. of Machine Learning and Systems*, volume 4, pages 33–51, 2022.

[39] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, Santa Clara, CA, February 2020. USENIX Association.

[40] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *USENIX Annual Technical Conference (ATC'21)*, pages 537–550, 2021.

[41] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–15, 2014.

[42] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *IEEE International Conference on Computer Vision (ICCV)*, pages 2999–3007, 2017.

[43] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In *Proceedings of ECCV*, 2014.

[44] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, February 2020.

[45] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *USENIX Annual Technical Conference (USENIX ATC 20)*, pages 189–203, 2020.

[46] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216. USENIX Association, February 2021.

[47] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. In *VLDB 2021*, January 2021.

[48] Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. In *VLDB 2021*, volume 14, 2021.

[49] Daniel S. Park and William Chan. SpecAugment: A New Data Augmentation Method for Automatic Speech Recognition. https://ai.googleblog.com/2019/04/specaugment-new-data-augmentation.html, 2019.

[50] Daniel S. Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D. Cubuk, and Quoc V. Le. Specaugment: A simple data augmentation method for automatic speech recognition. *Interspeech 2019*, Sep 2019.

[51] Lana Ramjit, Matteo Interlandi, Eugene Wu, and Ravi Netravali. Acorn: Aggressive result caching in distributed data processing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'19, page 206–219, 2019.

[52] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, 2016.

[53] Red Hat. Gluster: scalable data filesystem. https://www.gluster.org/, 2021.

[54] Red Hat. Setting up GlusterFS Volumes. https://docs.gluster.org/en/v3/Administrator%20Guide/Setting%20Up%20Volumes/, 2021.

[55] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, April 2021.

[56] Amazon Web Services. Amazon Simple Storage Service. https://aws.amazon.com/s3, 2021.

[57] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *J. Big Data*, 6:60, 2019.

[58] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of ICDAR*, page 958. IEEE Computer Society, 2003.

[59] Mingxing Tan and Quoc Le. Efficientnetv2: Smaller models and faster training. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pages 10096–10106. PMLR, 2021.

[60] TensorFlow. TensorFlow Graph Optimizations. https://research.google/pubs/pub48051.pdf, 2019.

[61] Tensorflow. Better performance with the tf.data API. https://www.tensorflow.org/guide/data_performance#caching, 2021.

[62] Tensorflow. tf.data.experimental.snapshot. https://www.tensorflow.org/api_docs/python/tf/data/experimental/snapshot, 2021.

[63] Jason Van Hulse, Taghi M. Khoshgoftaar, and Amri Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, page 935–942, 2007.

[64] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Liping Zhang Yong Li, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, April 2022.

[65] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, October 2018.

[66] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya Parameswaran. Helix: Accelerating human-in-the-loop machine learning. *Proc. VLDB Endow.*, 11(12):1958–1961, August 2018.

[67] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *International Conference on Computer Vision (ICCV)*, 2019.

[68] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021*, 2021.

[69] 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. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proc. of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, 2022.

[70] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *Proceedings of ICLR*, 2017.

A Artifact Evaluation README

A.1 Abstract

The artifact consists of the source code of Cachew[2], the Cachew client binaries[3], as well as scripts for building wheel files and Docker images. We also provide reference scripts for deploying GCE VMs for evaluation and for running the some representative experiments[4]. Do note that these scripts might not work as they depend on resources that might not be public. In these cases, experiment VMs will have to be manually set up.

The evaluation focuses on reproducing key experiments and their respective results which demonstrate how the main contributions of Cachew work:

- **Autoscaling** (*Figure 6a, compute curve):* show how input pipeline resources affect training time, how Cachew's autoscaling policy finds the right number of workers automatically, and how the Kubernetes Horizontal Pod Autoscaler fails to find the right scale.
- **Autocaching** (*Figure 8):* Show how Cachew's autocaching policy behaves under various execution scenarios, and how the most efficient execution mode is selected by Cachew.
- **Multi-tenancy with autoscaling and autocaching** (*Figure 11):* Show how Cachew behaves in multi-worker scenarios, selecting the most efficient execution mode, as well as the right scale for each job. Furthermore this experiment should also show how caching can be used in cross-job settings.

A.2 Artifact check-list

The artifact with the components listed below is available at: https://github.com/eth-easl/cachew_experiments.

- System to deploy: Cachew service (dispatcher, input data workers, remote cache cluster)
- Algorithms to evaluate: Cachew's autoscaling and autocaching policies
- Workloads to run:
  - *Figure 6a*: ResNet50 model and its open-source canonical input pipeline.
  - *Figure 8*: Synthetic input pipeline
  - *Figure 11*: Canonical ResNet50 input pipeline
- Binary: Cachew Docker image for workers and dispatcher, Cachew wheel file for client, GCE VM image.
- Model: ResNet50 and its canonical input pipeline
- Data Sets: ImageNet 2012 (stored in GCS bucket)
- Output: CSV files with metrics, text-based logs, and plots to compare with figures in the paper.
- Experiments: Experiments and deployment are fully scripted. See §A.4.
- Publicly available?: yes
- Code licenses: Apache 2.0
- Data licenses: ImageNet 3-Clause License
- Archived (provide DOI)? 10.5281/zenodo.6543943

A.3 Prerequisites

Hardware dependencies: The experiments require a cluster of x86 CPU servers with hardware virtualization support, with 4 Nvidia V100 GPUs on one of the servers. We recommend (and our scripts assume that you are) conducting experiments on Google Cloud. Some of the VM deployment scripts might not work out of the box as they can require access to resources which are no longer private. In this case, the scripts will either have to be modified or the deployment will have to be done manually.

Software dependencies: Our scripts make extensive use of the gcloud CLI tool. As a consequence, this tool is a prerequisite for setting up VMs and running experiments. Please follow this tutorial to install it. We additionally make use of the gsutil tool. To install it, please follow this tutorial. We also suggest to use Python 3.9 with PyEnv as a means to install and manage multiple python versions and virtual environments. The software requirements for the Google Cloud service deployment are installed on the VM images we provide.

Estimated time and cost: The estimated time needed to prepare the workflow is 30 minutes. The estimated execution time of experiments is approx. 15 hours in total. We provide an estimated breakdown of the time and cost for each experiment[5].

A.4 Instructions

Detailed instructions are provided in the artifact repository README. The evaluator will need to `git clone https://github.com/eth-easl/cachew_experiments.git` locally, then use the scripts provided in the `deploy` folder to spin up a VM, and later tear it down. Once the VM is spun up, one needs to `ssh` into the VM and use the scripts in the relevant experiment directory. Once the experiment is finished, the VM can be torn down. Note that as mentioned before, some of the scripts might not work due to private dependencies. In such cases, use the scripts as reference.

[2] https://github.com/eth-easl/cachew
[3] gs://cachew-builds/tensorflow-2.8.0-cp39-cp39-linux_x86_64.whl
[4] https://github.com/eth-easl/cachew_experiments
[5] Time and cost estimate sheet: https://tinyurl.com/52mwtccn

A.4.1 Getting Started

Please see the artifact repository Getting Started section of the README for instructions on how to write a simple Cachew input data pipeline and execute it locally.

A.4.2 Reproducing Experiment Results

We provide scripts to automate the deployment, execution, and result plotting for the three key experiments listed in §A.1. See the Artifact Evaluation section of the README for instructions to run the scripts. Please follow the following steps for each experiment:

1. Deploy a VM for artifact evaluation using `deploy/deploy.sh <vm-name> <gpu-count>`
2. Use the `gcloud compute ssh <vm-name>` command to `ssh` into the VM
3. Use `cd ${HOME}/cachew_experiments/experiments/<ename>` where `<ename>` is the experiment name, and follow the README there and the associated scripts to run the experiments.
4. Use `gcloud compute scp` to collect whatever resource you find relevant after the experiment is done.
5. Exit the `ssh` session, and tear down the VM using the `deploy/terminate.sh` script.

For Figure 6a (compute curve) experiment, which required 4 GPUs, see the `experiments/autoscaling` directory. For Figure 8, see the `experiments/autocaching` directory. For Figure 11, see the `experiments/multi-tenancy` directory.

A.5 Evaluation and Expected Results

Each of the three experiments produces a plot which should be comparable with the associated plot in the paper.

A.5.1 Experiment Metrics

- *Figure 6a*: Epoch time in seconds, number of workers chosen by Cachew's autoscale policy and number of workers chosen by Kubernetes HPA
- *Figure 8*: Batch time, Cachew's autocache policy decisions
- *Figure 11*: Epoch time, autocache policy decision, autoscale policy decision

A.5.2 Expected Results and Possible Variations

- *Figure 6a*: Epoch time can vary depending on cloud conditions. Decay may or may not be more or less aggressive due to this. Consequently, autoscale decision might vary around 4 workers (at most ±1 worker). While it is rare, it can happen that the Kubernetes HPA scaling also changes from one worker to two.
- *Figure 8*: Epoch times might vary due to cloud conditions. Shape of curves should still be the same (although *compute* might change as it depends heavily on GCS). Note that the points on the curves, and Cachew's decisions are recorded separately (i.e. in different runs). Consequently, conditions might change, and metrics could potentially vary leading Cachew to make a seemingly 'wrong' decision at points where the three options have similar throughput. Otherwise, Cachew decision expected to follow lowest batch time option.
- *Figure 11*: Job 1's first two epochs are not always expected to converge during autoscale phase (as Cachew prefers to move to next epoch in those compute modes), but it could happen at 3 workers (both epochs). Job 1's third epoch expected to converge around 2 workers. Job 2 is expected to converge around 4 workers. Epoch times for Job 1 should be around [366s, 363s, 266s, 253s] while for Job 2 around 158s initially then around 129s in the later epochs. The expected sequence of execution modes for Job 1 is [PROFILE, PUT, GET] and for Job 2 is only GET. Both jobs can have epoch extensions towards the end of a run. A reasonable amount of variability in the worker count (±1 worker) and epoch times is expected. This experiment is expected to be the relatively volatile, and emphasis should be placed on epoch time convergence and the autocache decisions. On rare occasions the autoscaling decisions can converge to a wrong scale due to the short and synthetic nature of the experiment and the implicit noise this causes on the scaling metrics. As this is a short job, Cachew cannot correct the scale in good time. We have also recently identified a bug caused by the merge with the recent TensorFlow 2.8 codebase which affects our cache store. More detailed information regarding this experiment's expected outcome and variability can be found in its README.

A.5.3 Experiment customization

It should be possible to modify some of the parameters pertaining to each experiment. For instance, for the Figure 6a experiment, it is possible to change the number of workers across which clusters are deployed. These parameters can be changed in the experiment scripts themselves or for some as command line parameters. Please see the experiment scripts for further details.