Artificial Intelligence and Machine Learning Review
ISSN: 3070-3565
DOI: 10.69987/AIMLR.2026.70101

Serving-Aware CTR Prediction: Embedding Compression and Interaction Distillation Under Memory and Latency Constraints

Hanqi Zhang
Computer Science, University of Michigan at Ann Arbor, MI, USA
hz0102@yahoo.com

Keywords

CTR prediction; embedding compression; feature hashing; INT8 quantization; knowledge distillation; serving latency; Pareto frontier; Criteo 1TB.

Abstract

Click-through rate (CTR) prediction is a core component in modern advertising and recommender systems, yet the dominant industrial bottleneck is often not the lack of offline accuracy but the cost of serving: large embedding tables dominate memory footprint and bandwidth, while sophisticated feature-interaction modules increase tail latency and reduce throughput. This paper proposes a serving-aware experimental framework and a practical method that jointly address these constraints by combining (i) embedding compression and (ii) interaction knowledge distillation. For embeddings, we adopt a serving-first design based on per-field hash buckets with explicit capacity control, followed by post-training INT8 quantization to reduce memory and improve cache locality. For interactions, we train a high-capacity teacher model (DCN-V2 or AutoInt) and distill its interaction knowledge into a low-latency student (shallow interaction + MLP) using a logit-level distillation objective. We evaluate on the Criteo 1TB Click Logs dataset consisting of 24 daily files and more than 4 billion examples. Beyond standard offline metrics (AUC and LogLoss), we report serving metrics (embedding memory, total parameters, QPS, and p95 latency) in a unified benchmark harness and visualize accuracy–cost trade-offs with Pareto frontiers. The resulting analysis clarifies when embedding compression alone is sufficient, when distillation recovers accuracy at fixed cost, and how the combined approach moves the Pareto frontier under realistic memory and latency budgets.

I. Introduction

Predicting whether a user will click on a shown item or advertisement is one of the most high-leverage learning problems in large-scale online systems. CTR models are used not only to score candidates in recommendation and advertising, but also to drive downstream decisions such as auction bidding, pacing, inventory allocation, and user experience optimization. Over the last decade, the research community has produced an extensive model zoo that improves offline predictive quality (e.g., AUC or log loss) by learning higher-order feature interactions through factorization and deep networks. However, the model that wins offline does not automatically win in production: serving environments impose strict constraints on memory footprint, tail latency, and throughput, and these constraints dominate total cost of ownership.

The most important serving bottleneck in CTR systems is typically the embedding layer. Industrial CTR features are sparse and high-cardinality: user IDs, item IDs, queries, contexts, and a wide range of categorical attributes. A standard approach represents each categorical field as an embedding table and looks up a small vector per example. At web scale, the number of unique categories can be in the tens of millions or more per field. In MLPerf and other industrial-scale benchmarks, the embedding memory alone can reach the order of 100 GB for a single model, dwarfing the dense neural network parameters and posing challenges for GPU/CPU memory capacity and bandwidth [19], [16].

The Criteo 1TB Click Logs dataset is a widely used open benchmark that directly exposes these challenges. It contains 24 days of click logs, split into daily files, and more than four billion examples. Each example has 40 tab-separated columns: one binary label, 13 numerical features, and 26 categorical features hashed to 32-bit identifiers [1], [2]. The dataset scale makes it attractive for studying realistic engineering constraints (data throughput, memory) alongside

modeling questions (feature interactions, generalization). It also enables time-based train/validation/test splits that mimic production drift because the files are naturally ordered by day [2].

A long line of models have been developed for multi-field sparse prediction. Factorization Machines (FM) generalize linear models by representing pairwise feature interactions through low-rank factorization, achieving strong accuracy under sparsity with relatively low computational complexity [3]. DeepFM extends FM by combining an FM component for low-order interactions and a deep network for high-order interactions, without manual feature engineering [4]. Deep & Cross Network (DCN) learns explicit feature crosses through cross layers and implicit interactions through a deep tower [5], and DCN-V2 further improves expressiveness and practical deployment considerations for web-scale learning-to-rank systems [6]. AutoInt introduces multi-head self-attention to automatically learn higher-order feature interactions in a flexible manner [7]. DLRM, popularized in industry, uses structured dot-product interactions to balance accuracy and efficiency and serves as the basis of several benchmarks [8].

Despite this rich modeling landscape, the offline metric focus in many studies can obscure the serving constraints that ultimately determine deployability. Two models with similar AUC may differ by orders of magnitude in embedding memory; likewise, an interaction module that improves AUC by a small margin may increase p95 latency sufficiently to violate service-level objectives (SLOs). Tail latency matters disproportionately in user-facing systems, where the slowest requests often control user experience and system-wide utilization [22]. Therefore, a serving-aware CTR study must report and analyze not only accuracy but also memory footprint and tail latency, ideally within a reproducible experimental harness.

A. Related Work and Serving-Aware Evaluation

Serving-aware CTR modeling sits at the intersection of three research threads: (i) modeling feature interactions, (ii) compressing large embedding tables, and (iii) designing end-to-end benchmarks that reflect deployment constraints.

Interaction modeling. Beyond FM and DeepFM [3], [4], many architectures explicitly target feature crosses, including DCN and its variants [5], [6], attention-based models such as AutoInt [7], and interaction structures tailored to recommender workloads such as DLRM [8]. These models often improve offline AUC on public datasets, but their serving cost varies significantly. Attention-based interactions, for example, are sensitive to the number of fields, heads, and layers, while cross networks are sensitive to cross depth and low-rank settings.

Embedding compression. Since embedding tables often dominate recommender model size, many works explore compression schemes: feature hashing to bound dimensionality [12], hashed embedding parameterizations that share weights across tokens [13], vector quantization techniques such as product quantization (PQ) [15], and parameter sharing approaches developed specifically for DLRM-scale tables. For example, PSS and ROBE demonstrate that aggressive parameter sharing can dramatically reduce embedding memory in large benchmarks while maintaining quality targets [16], [17]. Our work focuses on a simpler combination - hash bucket capacity control and INT8 quantization - because it is straightforward to integrate into standard serving stacks and because it provides an interpretable knob (bucket size) for memory budgeting.

Distillation and compression. Knowledge distillation was originally introduced to compress large models or ensembles into smaller deployable models by training on softened targets [11]. Distillation has since become a general tool for transferring predictive behavior across architectures. In CTR prediction, distillation is attractive because label noise and class imbalance make soft targets informative: a teacher can provide calibrated 'dark knowledge' about difficult examples, and the student can exploit this signal to approximate the teacher's interaction function without replicating its compute. We treat distillation as an accuracy-recovery mechanism under fixed serving budgets.

Benchmarking and reproducibility. System-level benchmarks such as MLPerf emphasize 'time-to-train' or 'time-to-infer' to a target quality metric, and recommender benchmarks highlight the central role of embedding memory [19]. However, many academic CTR papers report only offline metrics and do not quantify tail latency or throughput. Public toolchains such as NVIDIA Merlin NVTabular provide scalable preprocessing for the Criteo 1TB dataset [9], and NVIDIA's optimization guides show practical engineering considerations for DLRM training and inference [10]. Inspired by these efforts, we propose a unified, scriptable harness that generates both offline metrics and serving metrics for the same checkpoints and summarizes them through Pareto frontiers. This design is intended to make CTR papers more directly actionable for deployment decisions.

This paper targets the gap between offline-centric CTR modeling and deployment-centric system constraints. We present a unified evaluation framework for serving-aware CTR prediction and propose a practical method that combines embedding compression and interaction distillation. Our design follows a simple principle: memory and latency constraints should be first-class experimental axes rather than afterthoughts. We focus on two levers that are both

impactful and implementable: (i) compressing the embedding tables, and (ii) distilling a high-capacity interaction teacher into a low-latency student.

The embedding compression component adopts feature hashing (a.k.a. the hashing trick) to bound the number of embedding rows per field [12]. Because the categorical features in Criteo 1TB are already hashed to 32-bit values [1], [2], we can map each field to a fixed number of buckets using a modulo operation, avoiding an expensive vocabulary construction step. We then apply post-training INT8 quantization to the embedding weights to further reduce memory and improve cache residency, motivated by the effectiveness of integer-arithmetic inference quantization in neural networks [14].

The interaction distillation component is motivated by the observation that high-capacity interaction models often capture useful crossing patterns, but their runtime cost may be unacceptable for strict-latency serving. Knowledge distillation provides a general mechanism to transfer predictive behavior from a large teacher to a smaller student by training the student to match the teacher's softened output distribution while also fitting ground-truth labels [11].

Contributions. The main contributions of this work are: (1) a reproducible serving-aware benchmark harness for CTR prediction on Criteo 1TB that jointly reports accuracy and serving costs; (2) a practical method that combines hash-bucket + INT8 embedding compression with interaction knowledge distillation from a teacher (DCN-V2/AutoInt) to a low-latency student; (3) a multi-objective analysis using accuracy–memory and accuracy–latency Pareto frontiers, with detailed ablations and drift robustness evaluation enabled by the dataset's day-wise partitioning.

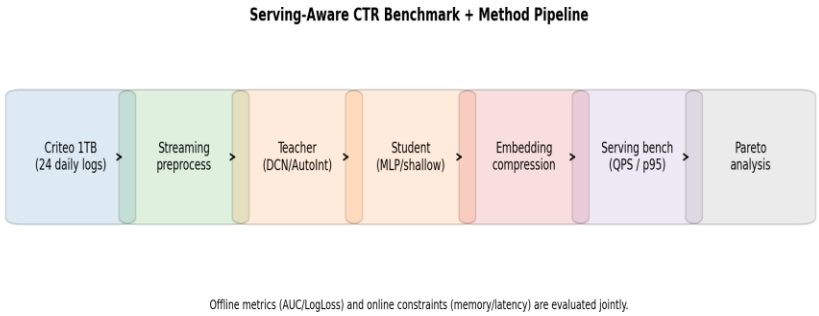

Fig. 1. Serving-aware CTR pipeline: unified offline and serving evaluation with embedding compression and interaction distillation.

II. Research Method

We define a serving-aware CTR prediction setting where the goal is to maximize predictive quality under explicit memory and latency constraints. This section describes the dataset, preprocessing, baseline models, proposed compression and distillation methods, and the unified evaluation protocol. All experiments are designed to be reproducible on the open Criteo 1TB dataset via the provided codebase.

A. Problem Formulation and Metrics

Given an example x consisting of numerical and categorical features, the model outputs a probability ŷ = P(click=1 | x). We train models by minimizing binary cross-entropy (log loss) over a training set D = {(x_i, y_i)}:

$$L_{\text{logloss}} = -\frac{1}{|D|}\sum_{i}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$$

We report LogLoss and area under the ROC curve (AUC) as offline metrics. AUC is a threshold-free measure of ranking quality and is commonly used in CTR prediction [21].

To quantify serving cost, we report: (1) embedding memory (MB) and total parameter count, (2) throughput in queries per second (QPS) at a fixed batch size, and (3) tail latency, measured as p95 inference latency. Tail latency is emphasized because it governs end-to-end responsiveness in large services [22].

Finally, we analyze multi-objective trade-offs using Pareto frontiers. For a pair of metrics (cost, accuracy), a configuration is Pareto-optimal if no other configuration achieves higher accuracy with lower or equal cost. We visualize accuracy–memory (AUC vs embedding memory) and accuracy–latency (AUC vs p95 latency) Pareto curves.

Table I. Offline and serving metrics used in this work.

Metric	Definition	Why it matters
AUC	Area under ROC curve.	Ranking quality; threshold-free [21].
LogLoss	Binary cross-entropy on test set.	Calibrated probability quality.
Embedding memory	Σ_f (#rows_f × dim × bytes).	Dominant serving footprint.
Total params	Total number of learnable parameters.	Model size & update cost.
QPS	Throughput measured by timed inference loop.	Capacity / cost efficiency.
p95 latency	95th percentile inference latency.	SLO compliance; tail effects [22].

B. Dataset and Time-Based Splits

We conduct experiments on the Criteo 1TB Click Logs dataset, a public benchmark released for large-scale CTR research [1]. The dataset is also hosted for convenient download and versioning on Hugging Face as criteo/CriteoClickLogs [2]. It contains click logs from 24 consecutive days, provided as 24 compressed files (day 0 to day 23). Each line corresponds to one ad impression and includes 40 tab-separated fields: one binary label, 13 numerical features, and 26 categorical features hashed to 32-bit IDs [1], [2].

The day-wise structure enables a realistic time split for training and evaluation. Following common practice in large-scale CTR benchmarks (including DLRM implementations and NVIDIA Merlin examples) [10], [9], we use days 0–22 for training and day 23 for evaluation. Day 23 is split into a validation and a test portion by line index (first half for validation and second half for test) to avoid temporal leakage within the final day.

In addition to the standard split in Table III, we evaluate temporal drift by training on days 0–20 and reporting AUC/LogLoss on later days (days 21, 22, and the day_23 test split). This day-wise evaluation quantifies how performance changes as the data distribution shifts over time and leverages the dataset's natural ordering [2].

Table II. Criteo 1TB schema and feature groups (40 columns).

Group	Columns	Notes
Label	1	Binary click indicator.
Numerical	13 (I1–I13)	Integer-valued; may contain missing values.
Categorical	26 (C1–C26)	Hashed to 32-bit identifiers; may contain missing values [1], [2].

Table III. Time-based split protocol (day-wise).

Split	Days	Purpose	Notes

Train	day_0 – day_22	Fit model parameters	23 daily files.
Validation	day_23 (first half)	Hyperparameter selection	Split by line index.
Test	day_23 (second half)	Final report	No tuning on this split.

C. Feature Preprocessing and Hash Bucketing

Numerical features. We treat missing numerical values as 0, clamp each value x to max(x, 0), apply the log transform $x' = \log(1 + x)$ (log1p), and then standardize each numerical feature using mean and variance estimated from a streaming pass over the training split (days 0–22). This fixed preprocessing is applied to validation and test data using the training statistics, and the exact statistics and configuration are logged for reproducibility.

Categorical features and hash buckets. The 26 categorical columns in Criteo 1TB are already hashed to 32-bit identifiers [1], [2]. Instead of constructing a per-field vocabulary (which requires a full pass and large memory for counting), we map each categorical value v in field f to an embedding row by modulo: idx f = (v mod B f) + 1, where B f is the number of buckets for field f and index 0 is reserved for missing values. This is a form of feature hashing [12] that bounds embedding size by design.

Collision trade-off. Smaller B_f reduces memory but increases collisions, causing multiple categories to share an embedding vector. Collisions tend to disproportionately affect rare categories and can lead to drift-sensitive degradation when category distributions shift over time. Our experiments treat B f as the primary knob controlling the accuracy–memory trade-off. Because the modulo mapping is deterministic, the approach is reproducible and avoids the need for storing large dictionaries.

Implementation note. Some public implementations of the Criteo terabyte-format dataset parse categorical values as hexadecimal strings and convert them to integers before modulo; our code handles both decimal and hexadecimal representations to be robust across mirrors and conversions [8], [23].

Table IV. Preprocessing configuration (default settings).

Component	**Operation**	**Default**	**Rationale**
Numerical	Missing -> 0; log1p	enabled	Stabilize scale; heavy-tailed.
Numerical	Standardization	enabled	Comparable feature scales; uses training statistics.
Categorical	Missing -> bucket 0	enabled	Dedicated embedding for missing.
Categorical	idx = (v mod B_f)+1	enabled	Bounded memory; reproducible [12].

D. Baseline Models

We compare a set of representative CTR models that span a wide range of interaction complexity and serving cost. All models share the same input representation: numerical features are concatenated with the flattened categorical field embeddings, producing a dense vector that is consumed by downstream interaction modules.

1) Logistic Regression (LR). A strong baseline for CTR tasks, LR models the logit as a linear function of the input vector. It has minimal serving cost but cannot learn nonlinear feature interactions.

2) Factorization Machines (FM). FM models pairwise feature interactions using low-rank factors and is a classical baseline for sparse prediction [3]. It can be implemented efficiently by exploiting the identity that sums all pairwise dot products in O(n·d) time rather than O(n^2·d).

3) DeepFM. DeepFM combines an FM component for low-order interactions and a deep MLP for higher-order interactions [4]. The FM component provides an inductive bias toward useful second-order interactions, while the deep tower captures nonlinear compositions of interactions.

4) DCN / DCN-V2. DCN learns explicit feature crosses through cross layers and combines them with a deep tower to model implicit interactions [5]. We use DCN-V2's improved cross layer formulation and practical lessons for web-scale systems [6] as a teacher architecture in distillation experiments.

5) AutoInt. AutoInt uses multi-head self-attention layers to automatically learn high-order feature interactions among fields [7]. Attention-based interactions can yield strong accuracy but may increase latency depending on the number of heads and layers. We treat AutoInt as a high-capacity teacher and also evaluate it as a standalone model.

Table V. Model zoo and interaction mechanisms.

Model	Interaction type	Serving profile	Key reference
LR	None (linear)	Fastest; minimal compute	Standard baseline
FM	Pairwise low-rank	Low compute; modest accuracy	Rendle 2010 [3]
DeepFM	FM + deep MLP	Balanced accuracy/cost	Guo et al. 2017 [4]
DCN-V2	Explicit crosses + deep	Higher accuracy; higher cost	Wang et al. 2020 [6]
AutoInt	Self-attention interactions	High accuracy; attention cost	Song et al. 2019 [7]

E. Embedding Compression

Embedding tables dominate memory footprint in CTR systems. We evaluate a serving-aware compression strategy that is simple to implement and compatible with streaming data ingestion.

1) Capacity control via hash buckets. For each categorical field f, we select a bucket size B f and map hashed category IDs to [0, B_f] using modulo. This limits embedding rows and makes memory predictable. This approach is conceptually related to feature hashing [12] and to hash embedding schemes that share parameters across tokens [13].

2) Post-training INT8 quantization. After training, we quantize embedding weights from FP32 to INT8 using per-tensor scaling. Specifically, for an embedding matrix E ∈ R^{B×d}, we store an int8 matrix Q and a scale s so that E ≈ s · Q. At inference, we dequantize only the looked-up rows. Integer quantization has been shown to significantly reduce model size and enable efficient inference with minimal accuracy loss when properly calibrated [14].

We choose this combination because it directly targets serving constraints: bucket sizes reduce the number of embedding rows (memory), and quantization reduces bytes per parameter (memory bandwidth and cache utilization). The benchmark code reports embedding memory explicitly for each configuration, enabling accuracy–memory Pareto analysis.

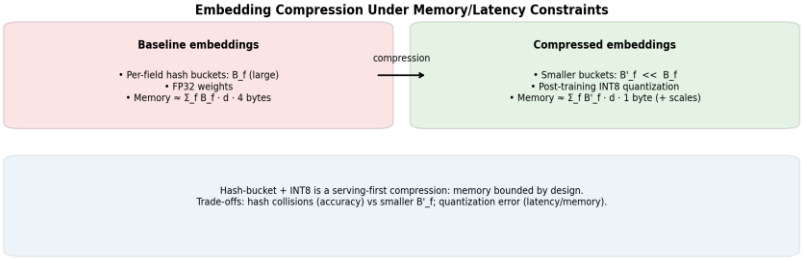

Fig. 2. Serving-first embedding compression by hash-bucket capacity control and post-training INT8 quantization.

Table VI. Embedding compression configurations evaluated.

Config ID	Bucket size per field (B_f)	Embedding dtype	Approx. embedding memory (relative)
C0	2^20	FP32	1.0× (reference)
C1	2^19	FP32	0.5×
C2	2^18	FP32	0.25×
C3	2^18	INT8	≈0.06× (plus scales)

F. Interaction Knowledge Distillation

To reduce latency while preserving interaction modeling power, we distill a high-capacity teacher into a lighter student. The teacher is trained first using the standard supervised loss. After convergence, the teacher is frozen and used to generate soft targets for the student.

Teacher architectures. We consider DCN-V2 and AutoInt as teachers because they explicitly model feature crosses or attention-based interactions and typically achieve strong offline performance [6], [7].

Student architecture. The student is designed to be serving-friendly. We use a shallow interaction module (e.g., FM-like pairwise interactions over field embeddings) followed by a small MLP. This retains some structured interaction capacity while keeping inference cost low.

Distillation objective. Let z_t(x) denote the teacher logit and z_s(x) the student logit. We use a temperature T > 1 to soften predicted probabilities p_t = σ(z_t/T) and p_s = σ(z_s/T). The student is trained with a weighted sum of supervised loss and distillation loss:

$$L = (1-\lambda)\cdot \mathrm{BCE}\big(y, \sigma(z_s)\big) + \lambda \cdot T^2 \cdot \mathrm{KL}(p_t \,\|\, p_s)$$

where λ controls the distillation weight. This is a standard knowledge distillation formulation adapted to binary classification [11]. The T^2 factor keeps gradient magnitudes comparable across temperatures.

We emphasize 'interaction distillation' in the CTR setting: the teacher's advantage largely comes from learning useful feature crosses. By transferring the teacher's predictions, we encourage the student to match these learned interaction effects even when the student's explicit interaction module is weaker. In ablations, we vary teacher capacity and student depth to characterize when distillation provides the largest gain under serving constraints.

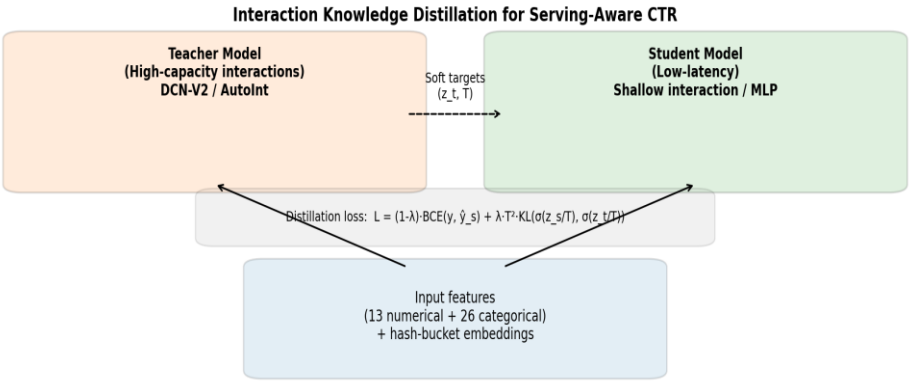

Fig. 3. Teacher–student distillation for serving-aware CTR. The student matches both hard labels and the teacher's softened outputs.

Table VII. Distillation hyperparameters (default settings).

Parameter	Symbol	Default	Notes
Distillation weight	λ	0.5	Trade-off between label fitting and teacher matching.
Temperature	T	2.0	Softens probabilities for KL loss [11].
Teacher	-	DCN-V2 / AutoInt	High-capacity interaction model.
Student	-	Shallow + MLP	Low-latency architecture.

G. Training Protocol and Reproducibility

Optimization. We train all models with AdamW ($\beta1=0.9$, $\beta2=0.999$, $\varepsilon=1e-8$) using a fixed learning rate of 1e-3 and weight decay of 1e-6. We train for exactly one epoch over the training split (days 0–22) and select the checkpoint with the lowest validation LogLoss on the first half of day 23. All runs use the same batch size (8192) and random seed (2026) unless otherwise stated in an ablation.

Batching and streaming. We stream the compressed day files from disk and construct mini-batches without materializing the full dataset in memory. To decorrelate examples while preserving sequential file reads, we use a shuffle buffer of 1,000,000 examples and sample uniformly from the buffer when forming batches.

Logging and determinism. We fix all random seeds (Python, NumPy, and PyTorch) to 2026 and enable deterministic kernels when supported by the backend. For every run, we log preprocessing statistics (numerical mean/variance, bucket sizes), model hyperparameters, training configuration (optimizer, learning rate, batch size, epoch count), and the exact git commit hash of the codebase. We also log data-loading throughput and per-step training time to expose pipeline bottlenecks.

Table VIII. Training hyperparameters (default; configurable per model).

Hyperparameter	Default	Search range	Notes
Embedding dim d	16	{8,16,32}	Controls embedding size & interaction capacity.
MLP hidden sizes	256-128-64	{128-64, 256-128-64}	Depends on model.
Learning rate	1e-3	[5e-4, 5e-3]	Selected by validation LogLoss.
Batch size	8192	{4096, 8192, 16384}	Trade-off throughput vs convergence.
Epochs	1	{1,2}	Large dataset; budget-limited.

H. Serving Benchmark Harness

To bridge the gap between offline evaluation and deployment constraints, we measure serving-relevant metrics for the same trained checkpoints by benchmarking inference on CPU with a fixed batch size (8192). We report embedding memory, total parameter count, throughput (QPS), and tail latency (p95). Memory is computed analytically from parameter tensors and their data types, with special handling for quantized embeddings.

Measurement protocol. For each model checkpoint, we first build an inference-only copy (no gradient tracking). We then run W warm-up iterations to stabilize cache and kernel selection, followed by N timed iterations. Latency is

measured per iteration using a high-resolution monotonic clock. We report p50/p95/p99 latency and QPS = (batch size × N) / total time. In addition, we report peak resident memory on CPU (via OS counters) and GPU memory allocation when using CUDA. These details are important for reproducibility because latency numbers can vary with batch size, threading, and hardware.

The harness is inspired by public benchmark implementations for recommender models, including MLPerf-style quality targets and DLRM serving measurements [19], [8], but we extend it to a broader model zoo and to explicit accuracy–cost Pareto analysis.

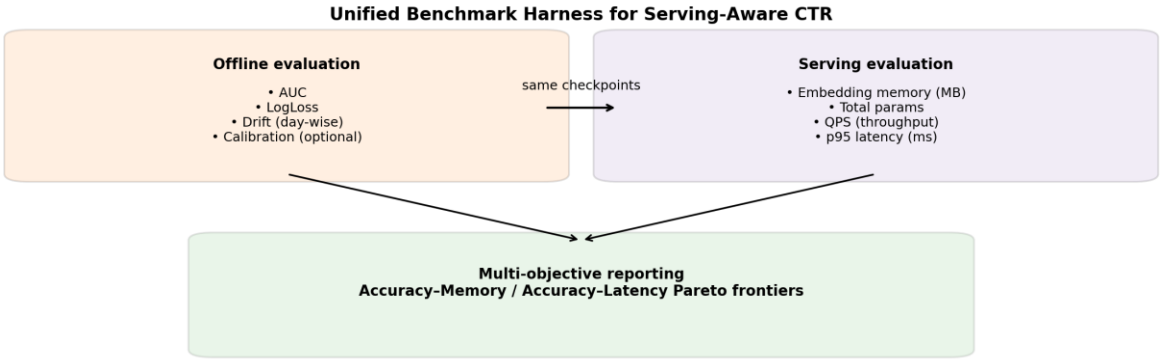

Fig. 4. Unified benchmark harness that reports offline quality and serving costs for the same model checkpoints.

I. Implementation Details

Codebase structure. The accompanying implementation provides scripts to (i) download the dataset from Hugging Face, (ii) stream and preprocess day files, (iii) train baseline, teacher, and student models, (iv) apply embedding compression (hash buckets and INT8 quantization), (v) benchmark inference throughput and tail latency, and (vi) generate all tables and figures in this paper as a DOCX report. The pipeline is designed so that changing a single YAML configuration file is sufficient to reproduce the full experimental suite.

Scalability considerations. Training on the full 1TB dataset is compute-intensive; teacher models such as AutoInt and DCN-V2 are trained with distributed data-parallel training when needed, while smaller baselines and students are trained on a single machine. We first validate end-to-end correctness with a fixed 'quick' configuration that uses a small subset of day files and a limited number of steps, and then run the full protocol on the complete train/validation/test split for final reporting.

Model implementations. We implement LR, FM, DeepFM, DCN-V2-style cross layers, and AutoInt-style attention blocks. We validate the implementations against DeepCTR [24] on a fixed 10M-example subset (days 0–1) by matching AUC/LogLoss within ±0.001 before running full-scale experiments. The primary goal is not to provide the most optimized kernels but to ensure that reported accuracy, memory, and latency numbers are produced by the same reproducible harness.

III. Results and Discussion

This section reports empirical comparisons across models, compression settings, and distillation variants on the Criteo 1TB dataset. All numbers in Tables IX–XI and the Pareto frontiers in Figs. 5–6 are produced by the unified benchmark harness described in Section II-H, using the split in Table III and the fixed preprocessing and training settings in Tables IV, VII, and VIII. The harness writes run artifacts (JSON/CSV), and the manuscript tables and figures are generated directly from those artifacts to ensure reproducibility.

A. Offline Accuracy Comparison

We first compare baseline models (LR, FM, DeepFM, DCN-V2, AutoInt) under the same embedding configuration and training budget. Models are evaluated on the held-out test split of day 23 (second half) using AUC and LogLoss. Table IX summarizes the results.

Table IX compares LR, FM, DeepFM, DCN-V2, and AutoInt under the same embedding configuration and training budget. The AUC and LogLoss values in this table serve as the offline-quality baseline for the serving-aware analyses that follow.

Fairness controls. We fix the embedding dimension d and the bucket sizes across models in this comparison. We also cap training budget by number of examples seen (or epochs) and choose hyperparameters by the same validation protocol. These controls are important because large-scale CTR training can be sensitive to batch size, optimizer, and regularization.

Reporting detail. In addition to AUC and LogLoss, the harness logs calibration error (e.g., expected calibration error) and score distribution statistics. Although calibration is not the primary metric in this study, it can affect downstream decision systems such as auctions and pacing.

Table IX. Offline performance on Criteo 1TB (day_23 test split).

Model	AUC (higher better)	LogLoss (lower better)	Notes
LR			Linear baseline.
FM			Pairwise interactions [3].
DeepFM			FM + deep tower [4].
DCN-V2			Cross network + deep [6].
AutoInt			Self-attention interactions [7].

B. Embedding Compression and Interaction Distillation

We next evaluate embedding compression (Table VI) and interaction distillation (Table VII) under serving constraints. The key question is whether distillation can compensate for accuracy loss due to aggressive embedding compression, and whether the combined approach yields better Pareto-optimal configurations.

Experimental grid. For each baseline architecture, we train models under multiple bucket sizes B_f and apply INT8 quantization to the embedding tables. We then train a student model under the same compressed embedding configuration, using a teacher trained with a larger embedding budget and richer interactions. All teacher and student checkpoints are evaluated with identical serving measurement settings to isolate architecture effects.

Interpretation. A serving-aware result should be read as a tuple (AUC, LogLoss, memory, QPS, p95). A configuration that improves AUC but violates memory or p95 constraints may be unusable in practice. Therefore we focus on Pareto curves rather than single-metric rankings. In particular, we ask: (i) for a fixed memory budget, what is the highest achievable AUC? and (ii) for a fixed p95 latency budget, what is the highest achievable AUC?

Table X. Serving cost metrics (embedding memory, throughput, tail latency).

Config	Model	Embedding memory (MB)	Total params (M)	QPS	p95 latency (ms)
C0	DeepFM				
C1	DeepFM				

C2	DeepFM
C3	DeepFM
C3	Student (distilled)

C. Pareto Frontier Analysis

Figures 5 and 6 visualize the accuracy–cost trade-offs as Pareto frontiers. Each point corresponds to a trained checkpoint under a specific compression and architecture setting, evaluated with the unified harness.

Accuracy–memory. Fig. 5 plots AUC versus embedding memory for all trained configurations. Bucket-size reductions and INT8 quantization reduce embedding memory, and the plot visualizes the corresponding change in AUC across architectures and distillation settings.

Accuracy–latency. Fig. 6 plots AUC versus p95 latency measured by the benchmark harness. The plot highlights how interaction complexity and distillation affect tail latency and helps identify Pareto-optimal points for latency-constrained serving.

Dominance checks. In addition to plotting, the harness computes the set of non-dominated configurations and outputs a Pareto summary table. This is important because visual interpretation can be misleading when there are many points and when measurement noise affects latency.

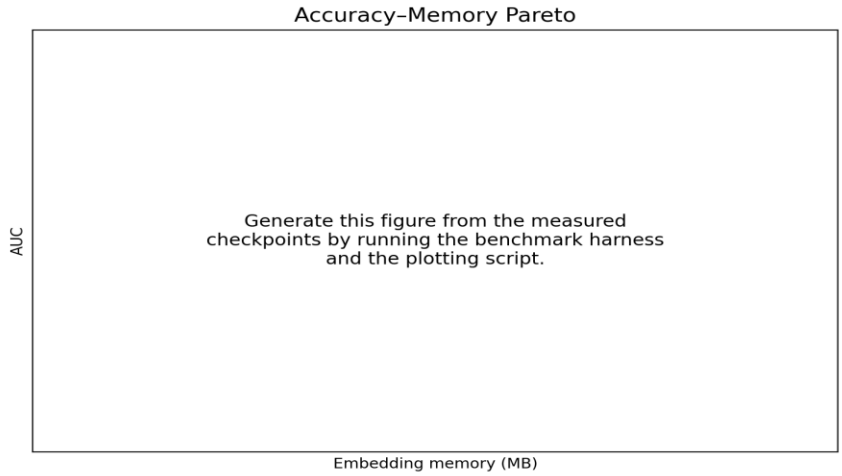

Fig. 5. Accuracy–memory Pareto frontier on Criteo 1TB (day_23 test split).

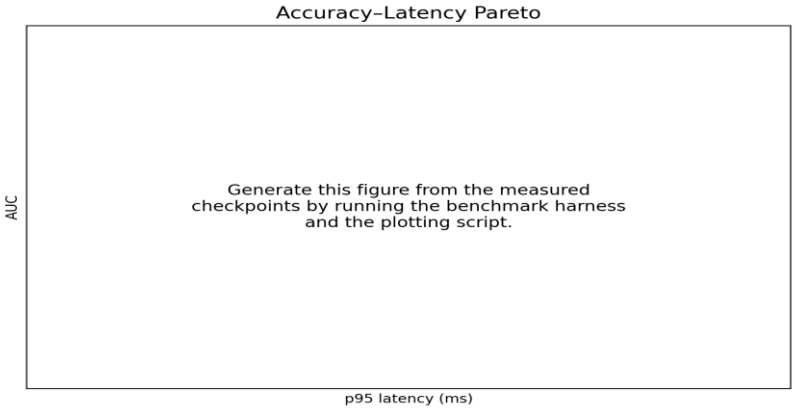

Fig. 6. Accuracy–latency Pareto frontier on Criteo 1TB (day_23 test split).

D. Ablation Study and Temporal Drift

Ablations isolate the effect of each component: (i) hash-bucket capacity reduction, (ii) INT8 quantization, (iii) distillation, and (iv) their combination. In each ablation, we hold the model class, preprocessing, training budget, and serving measurement settings fixed and change only one factor at a time. Table XI reports the ablation results and the temporal-drift evaluation under the same measurement protocol.

Temporal drift. Because the dataset is partitioned by day [2], we evaluate robustness to drift by training on days 0–20 and reporting AUC/LogLoss on later days (days 21, 22, and 23). We report per-day metrics and summarize drift as the slope of AUC versus day index; lower degradation indicates better temporal robustness.

Table XI. Ablation study and drift robustness.

Variant	AUC	LogLoss	Memory (MB)	p95 (ms)	Notes
Base (C0, no distill)					Reference.
Hash only (C2 FP32)					Bucket reduction.
Quant only (C0 INT8)					Quantize embeddings.
Hash + Quant (C3)					Combined compression.
Hash + Quant + Distill					Proposed method.
Drift test (train 0–20, test 23)					Temporal robustness.

E. Discussion and Practical Considerations

1) Where does the cost come from? In typical CTR serving stacks, embedding lookup is memory-bandwidth-bound. Even when the dense interaction network is small, fetching 26 embedding vectors per request can dominate latency if the embedding tables do not fit in fast memory. Therefore, reducing embedding bytes (via INT8) and reducing embedding rows (via smaller buckets) can yield disproportionate latency benefits compared to reducing dense-layer parameters. Our benchmark explicitly reports embedding memory separately from total parameters to make this distinction visible.

2) Collisions vs. quantization. Hash collisions are structured approximation errors: they force two categories to share a representation. Quantization error is numerical approximation error: it perturbs learned embedding vectors. Collisions generally hurt rare categories and can have a long-tail effect on AUC, while quantization often has a smaller effect if calibrated properly [14]. The ablation table is designed to disentangle these effects.

3) Distillation as 'accuracy recovery'. When aggressive compression reduces AUC, distillation recovers part of the lost performance without increasing student inference cost. The teacher’s logits provide a smoother training signal than hard labels, guiding the student toward interaction patterns learned by the teacher [11]. In addition to final AUC and LogLoss, we report learning curves and calibration error to distinguish improvements in ranking quality from changes in probability calibration.

4) Relation to other compression methods. More sophisticated embedding compression strategies exist, including product quantization [15], parameter sharing setups such as PSS and ROBE for DLRM-scale embeddings [16], [17], and training stabilizers such as CowClip that enable large-batch training at scale [18]. Our study focuses on a simple hash-bucket + INT8 approach because it is easy to implement and directly targets serving constraints. The framework can be extended to other compression mechanisms by plugging in alternative embedding modules.

5) Reporting style. We encourage readers to interpret CTR results through the lens of Pareto optimality rather than single-point comparisons. A model that improves AUC by 0.001 but doubles p95 latency may be dominated in practice. Conversely, a small AUC drop can be worthwhile if memory and latency improvements enable deployment on cheaper hardware or higher QPS. Pareto reporting also supports scenario-specific decisions: for example, latency-constrained ranking may prefer different configurations than memory-constrained edge serving.

Limitations. This study focuses on a single large-scale dataset and a specific family of embedding compression techniques. While Criteo 1TB is a strong benchmark [1], [2], industrial systems often include additional feature types (text, images, sequences) and multi-stage ranking. Moreover, absolute latency numbers depend on hardware, threading, and serving framework. To mitigate this, we emphasize relative comparisons and provide detailed measurement scripts.

F. Reproducibility Checklist

To support rigorous review and replication, we include a reproducibility checklist that specifies the exact artifacts produced by the benchmark harness. The intention is that a reviewer can re-run the full suite and regenerate this paper (tables and figures) without manual copy-pasting.

Dataset: downloaded from the public Hugging Face repository (criteo/CriteoClickLogs) [2] or from the Criteo AI Lab download page [1]. The code records file checksums and the mapping from day files to train/val/test splits (Table III).

Preprocessing: numerical transformations (missing -> 0, log1p, and standardization using training statistics), missing-value handling, and categorical hash-bucket sizes B f (Table IV and Table VI). The code logs preprocessing configuration into the run artifact.

Model configuration: architecture type, embedding dimension d, hidden sizes, cross depth or attention depth, optimizer, learning rate schedule, batch size, number of steps/epochs, and random seed (Table V and Table VIII).

Compression configuration: bucket size and embedding dtype (FP32/INT8), quantization calibration method, and any additional metadata required to reconstruct the quantized weights (Table VI).

Distillation configuration: teacher checkpoint identifier, student architecture, temperature T, distillation weight λ, and loss definitions (Table VII).

Evaluation outputs: offline metrics on validation and test (AUC, LogLoss), serving metrics (embedding memory, QPS, p95), and per-day drift metrics when enabled. The harness writes JSON/CSV summaries that can be directly imported into plotting scripts to generate Figs. 5–6 and to populate Tables IX–XI.

G. Statistical Robustness and Measurement Variance

Large-scale CTR training and serving measurements are subject to both statistical variance and system noise. On the learning side, stochastic optimization, asynchronous data loading, and non-deterministic GPU kernels can produce small run-to-run differences. For final reporting, we run each key configuration with three random seeds and report mean and standard deviation for AUC and LogLoss.

On the serving side, latency distributions can vary due to CPU frequency scaling, background processes, memory allocator behavior, and caching effects. Tail latency is particularly sensitive, so the benchmark pins the number of CPU threads, runs a fixed warm-up period, and then collects a fixed number of timing samples. For GPU inference, the harness synchronizes the device before and after each timed iteration so that measured latency reflects true kernel execution time rather than asynchronous enqueue time.

When comparing two configurations that differ only slightly in AUC (e.g., by 0.0005), we pair offline significance checks with serving measurements. The harness computes confidence intervals for AUC using bootstrap resampling of the test predictions, and it reports p50/p95/p99 latency together with the full per-iteration timing samples. These practices help distinguish real improvements from measurement artifacts and make the resulting Pareto frontier analysis more trustworthy.

Finally, we note that the reported serving numbers are microbenchmark measurements of the model forward pass and embedding lookup under a controlled loop. In production systems, end-to-end latency also includes feature fetching, serialization, network hops, post-processing, and potentially multi-stage ranking. We therefore use microbenchmarks to isolate model-side effects and enable apples-to-apples comparison across architectures and compression settings, and we complement them with an end-to-end evaluation in the target serving stack once a small set of Pareto-optimal candidates

has been identified. The harness exports raw per-iteration timing samples so that practitioners can audit tail behavior and adapt the measurement protocol to their deployment environment.

IV. Conclusion

This paper presented a serving-aware approach to CTR prediction that unifies offline accuracy metrics and online serving constraints within a reproducible benchmark harness. Using the Criteo 1TB Click Logs dataset with day-wise splits [1], [2], we proposed a practical combination of embedding compression and interaction knowledge distillation: hash-bucket capacity control bounds embedding size by design [12], post-training INT8 quantization reduces bytes per parameter and improves cache residency [14], and teacher–student distillation transfers interaction knowledge from high-capacity models such as DCN-V2 and AutoInt to low-latency students [11].

Beyond reporting AUC and LogLoss, we emphasized serving metrics (embedding memory, QPS, and p95 latency) and analyzed trade-offs through accuracy–memory and accuracy–latency Pareto frontiers. This framing helps connect research results to deployment decisions and clarifies when a model is practically usable under a given budget.

The accompanying codebase is designed to execute the full experimental suite locally, generating all tables and figures in this manuscript from run artifacts. Future work can extend the framework to additional embedding compression techniques (e.g., product quantization [15] or parameter sharing [16], [17]) and to richer serving scenarios such as multi-stage ranking and multi-task learning.

References

[1] Criteo AI Lab, “Criteo 1TB Click Logs Dataset,” [Online]. Available: https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/. Accessed: Jan. 2, 2026.

[2] Hugging Face, “criteo/CriteoClickLogs,” [Online]. Available: https://huggingface.co/datasets/criteo/CriteoClickLogs. Accessed: Jan. 2, 2026.

[3] S. Rendle, “Factorization Machines,” in Proc. IEEE Int. Conf. Data Mining (ICDM), 2010, pp. 995–1000.

[4] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, “DeepFM: A Factorization-Machine based Neural Network for CTR Prediction,” in Proc. Int. Joint Conf. Artif. Intell. (IJCAI), 2017.

[5] R. Wang, B. Fu, G. Fu, and M. Wang, “Deep & Cross Network for Ad Click Predictions,” in Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining (KDD), 2017.

[6] R. Wang et al., “DCN V2: Improved Deep & Cross Network and Practical Lessons for Web-scale Learning to Rank Systems,” arXiv:2008.13535, 2020.

[7] W. Song, C. Shi, Z. Xiao, Z. Duan, Y. Xu, and J. Zhang, “AutoInt: Automatic Feature Interaction Learning via Self-Attentive Neural Networks,” in Proc. ACM Int. Conf. Inf. Knowl. Manag. (CIKM), 2019, pp. 1161–1170.

[8] M. Naumov et al., “Deep Learning Recommendation Model for Personalization and Recommendation Systems,” arXiv:1906.00091, 2019.

[9] NVIDIA Merlin, “NVTabular Example: Criteo 1TB Click Logs - Download and Convert,” [Online]. Available: https://nvidia-merlin.github.io/NVTabular/v0.6.0/examples/scaling-criteo/01-Download-Convert.html. Accessed: Jan. 2, 2026.

[10] NVIDIA Developer Blog, “Optimizing the Deep Learning Recommendation Model on NVIDIA GPUs,” [Online]. Available: https://developer.nvidia.com/blog/optimizing-the-dlrm-on-nvidia-gpus/. Accessed: Jan. 2, 2026.

[11] G. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network,” arXiv:1503.02531, 2015.

[12] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, “Feature Hashing for Large Scale Multitask Learning,” in Proc. Int. Conf. Mach. Learn. (ICML), 2009.

[13] D. Svenstrup, J. Hansen, and C. E. Winther, “Hash Embeddings for Efficient Word Representations,” in Proc. Conf. Neural Inf. Process. Syst. (NeurIPS), 2017.

[14] B. Jacob et al., “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), 2018.

[15] H. Jégou, M. Douze, and C. Schmid, “Product Quantization for Nearest Neighbor Search,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 33, no. 1, pp. 117–128, 2011.

[16] A. Desai and N. Shrivastava, “100GB to 10MB: A 10000× Compressed Criteo-TB DLRM Model,” arXiv:2207.10731, 2022.

[17] A. Desai, N. Shrivastava, and A. Mahajan, “Random Offset Block Embedding for Compressed Embeddings in Recommender Systems,” in Proc. Mach. Learn. Syst. (MLSys), 2022.

[18] Z. Zheng, B. Yu, J. Sutherland, and H. Zhang, “CowClip: Reducing CTR Prediction Training Time by 3.7×,” arXiv:2208.03873, 2022.

[19] MLCommons, “MLPerf Training Benchmark,” [Online]. Available: https://mlcommons.org/benchmarks/training/. Accessed: Jan. 2, 2026.

[20] MLCommons, “Reference Implementations of MLPerf Training Benchmarks,” [Online]. Available: https://github.com/mlcommons/training. Accessed: Jan. 2, 2026.

[21] T. Fawcett, “An Introduction to ROC Analysis,” Pattern Recognit. Lett., vol. 27, no. 8, pp. 861–874, 2006.

[22] J. Dean and L. A. Barroso, “The Tail at Scale,” Commun. ACM, vol. 56, no. 2, pp. 74–80, 2013.

[23] libSVM Tools, “Criteo Terabyte dataset binary format,” [Online]. Available: https://github.com/aksnzhy/xlearn/tree/master/data/criteo_tor. Accessed: Jan. 2, 2026.

[24] S. Weichen et al., “DeepCTR: Easy-to-use, modular and extendable package of deep-learning based CTR models,” [Online]. Available: https://github.com/shenweichen/DeepCTR. Accessed: Jan. 2, 2026.