CloverRec: Cost-Efficient Disaggregated Processing-in-Memory Systems for Deep Recommendation Inference

Menglei Chen, Yu Hua*, Zhijun Yang, Yixiao Wang, Xumin Chen
Huazhong University of Science and Technology
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

Deep recommendation systems enhance user experiences by providing fast, high-quality personalized recommendations, which requires large-capacity and high-speed memory. While memory disaggregation offers a cost-efficient solution for handling large-scale embedding vectors (EVs), it suffers from significant network overheads. Processing-in-memory (PIM) technology restructures the data access path by offloading bandwidth-intensive embedding operations to remote PIM pools, hence minimizing data movement and alleviating network transfer costs. However, fully exploiting the bandwidth potential of disaggregated PIM pools is hindered by sub-optimal intra-unit parallelism, cross-unit load imbalance, and granularity mismatch in the host-device data transfer. To tackle these challenges, we propose CloverRec, a cost-efficient and high-performance recommendation inference system that offloads embedding operations to a disaggregated PIM pool. CloverRec employs a fine-grained dimension-partitioned data layout to achieve synchronization-free parallelism, introduces a dynamic data placement mechanism to mitigate hot-spot skewness across thousands of PIM units, and orchestrates hierarchy-aware data transfer at rank granularity to suppress transfer amplification and saturate the host-device bandwidth. Evaluations on a real UPMEM PIM system show that CloverRec outperforms state-of-the-art embedding schemes by up to 8.18×.

1 Introduction

Recommendation systems have become a fundamental component in many applications across various domains, such as e-commerce [62, 70], personalized advertisements [21, 23], and social media [58]. The core driving force behind recommendation systems lies in their capability to capture user behaviors and preferences while distilling essentials from vast amounts of information. Modern recommendation systems widely employ *deep learning recommendation models (DLRMs)* [1, 14, 65] to enhance the accuracy. Such deep recommendation systems efficiently handle billion-scale users and items while delivering substantial enhancements to user experiences.

Deep recommendation systems typically consist of two parts: an embedding layer that creates low-dimensional representations of input features, and a compute-intensive top multi-layer perceptron (MLP) that scores items. The inputs of deep recommendation systems are generally categorized into two types: dense features (e.g., product price) and sparse features (e.g., product ID) [1, 65]. Recommendation systems transform sparse features into dense representations by encoding them as embedding vectors (EVs) [3]. The trained EVs are stored in various EV tables according to their properties. To process an inference request, the deep recommendation systems retrieve the target EVs by performing lookups in the EV tables using the sparse features as keys. These EVs are grouped by their table IDs, forming several *EV groups*. The EVs within the same EV group are further reduced via pooling mechanisms (e.g., sum/max).

In contrast to the compute-intensive MLP layers, the embedding layer exhibits low compute intensity and demands high bandwidth. Enterprise-level deep recommendation systems typically adopt an *embedding-disaggregated structure* [14, 58] to enhance resource utilization and optimize cost-efficiency, where GPUs accelerate MLP layers and CPUs handle embedding layers. The embedding-disaggregated structure enables flexible CPU/GPU configurations, achieving higher resource efficiency.

To improve recommendation accuracy and support richer feature representations, recommendation systems tend to store more EVs with higher dimensions [28, 60], leading to an unprecedented scale of EV tables. Recent studies report that EV tables in enterprise-level recommendation systems will scale to dozens of terabytes [2, 49, 76], which cannot be fully placed into the memory of a single machine. Fully GPU-resident deployments are even less cost-efficient, since they over-provision expensive GPU memory for massive, sparse, and low-compute-intensity embedding tables. Therefore, practical deployments need remote embedding access to decouple embedding capacity from GPU compute capacity.

Memory disaggregation [26, 52, 53, 57, 63, 64, 78, 79] allows flexible resource management by decoupling the compute and

memory resources, which becomes a promising solution to provide high scalability and resource utilization in modern data centers. By adopting memory disaggregation schemes, recommendation systems can store EV tables in remote memory pools and independently scale them with high cost-efficiency. However, the recommendation systems need to fetch a large number of EVs from the remote memory pool during inference, ranging from tens to hundreds of EVs per table for one request [23, 56, 82], making GB-scale data movements for each inference. Therefore, despite using high-performance networks (e.g., RDMA [16] or CXL [11]) to connect different resource pools and caching hot EVs in the compute pools, the memory-disaggregated embedding scheme still suffers from significant data transfer overheads.

Due to the salient features of minimizing data movements and achieving high bandwidth and energy efficiency, processing-in-memory (PIM) [33, 34, 36–39, 42] offers a non-trivial new platform and performs computations closer to where data resides. PIM becomes an efficient solution to reduce the network overheads of memory-disaggregated embedding schemes. To this end, we explore offloading embedding operations to a disaggregated PIM pool, which significantly reduces network transfer costs. However, simply offloading embedding operations to real PIM systems unfortunately suffers from the following system-level challenges:

1) Inefficient intra-DPU parallelism. A PIM system consists of thousands of *DRAM processing units (DPUs)* [69], each of which handles embedding lookups and reduction of EV groups. Each DPU allows its threads to perform embedding operations for different EV groups. However, this group-based parallelism causes load imbalance among threads due to varying group sizes, resulting in low performance. While performing operations at the EV granularity can address the load imbalance issue, it introduces significant thread contention caused by lock-based correctness mechanisms, ultimately failing to achieve high performance.

2) Performance degradation due to inter-DPU load imbalance. A PIM system leverages thousands of DPUs to deliver high bandwidth and offer parallelism, making load balancing across DPUs crucial. However, real-world workloads often exhibit skewed access patterns, where a few hot EVs account for most accesses [1, 29, 35, 73], resulting in significant load imbalance across DPUs. Such a load imbalance deteriorates the performance.

3) High transfer amplification due to granularity mismatch in PIM processing. DPUs are generally managed at the *DPU-Set* (all allocated DPUs) granularity. However, the PIM system distribute and execute tasks at the granularity of a *single DPU*. Since the real UPMEM PIM product [13] requires transferring data of the *same* size in a single transmission due to data alignment, the granularity mismatch in PIM processing results in significant transfer amplification. Furthermore, all DPUs have to wait for the slowest one during processing, which decreases the system performance.

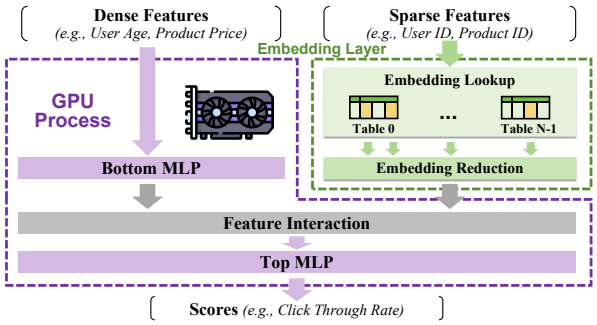


Figure 1: The workflow of a deep recommendation system.

To efficiently address the mentioned problems, we propose CloverRec, a PIM-enabled recommendation system that delivers cost-efficient inference. CloverRec fully leverages the high parallelism of DPU threads and achieves load balancing across DPUs, while further mitigating transfer amplification with rank-level PIM processing. Specifically, this paper makes the following contributions:

- ***Dimension parallelism.*** To efficiently utilize DPU threads, CloverRec redesigns the intra-unit data processing scheme via a fine-grained dimension parallelism, enabling lock-free concurrent access to distinct data slices partitioned by dimension ranges. This approach fully exploits the parallelism of DPU threads while minimizing inter-thread synchronization overheads, hence gaining high performance.
- ***Dynamic redundancy-based load balancing.*** To make thousands of DPUs load-balanced, CloverRec creates multiple replicas of hot EVs across DPUs. CloverRec leverages lightweight *split* and *merge* operations to dynamically adjust the replication scheme, hence enabling efficient adaptation to various access patterns. To reduce CPU computation costs, CloverRec facilitates intra-DPU reduction and proactively controls the redundancy degree (i.e., the number of replicas).
- ***Rank-level PIM processing.*** To alleviate transmission amplification and saturate CPU-DPU bandwidth, CloverRec implements a hierarchy-aware transfer mechanism that aligns data transfer granularity with the physical DIMM rank structure, minimizing padding overhead. CloverRec further minimizes transfer costs by pipelining data transmission with DPU execution.
- ***Real implementation and comprehensive evaluation.*** We have implemented and evaluated CloverRec on a real PIM system using the representative recommendation framework DLRM [14, 58]. Extensive experimental results show that CloverRec outperforms state-of-the-art embedding disaggregation schemes by up to 8.18×. We have released the open-source codes for public use in `https://github.com/LighT-chenml/CloverRec`.

2 Background and Motivation

2.1 Deep Recommendation Systems

As shown in Figure 1, a typical deep recommendation system processes two types of inputs: dense features (e.g., user

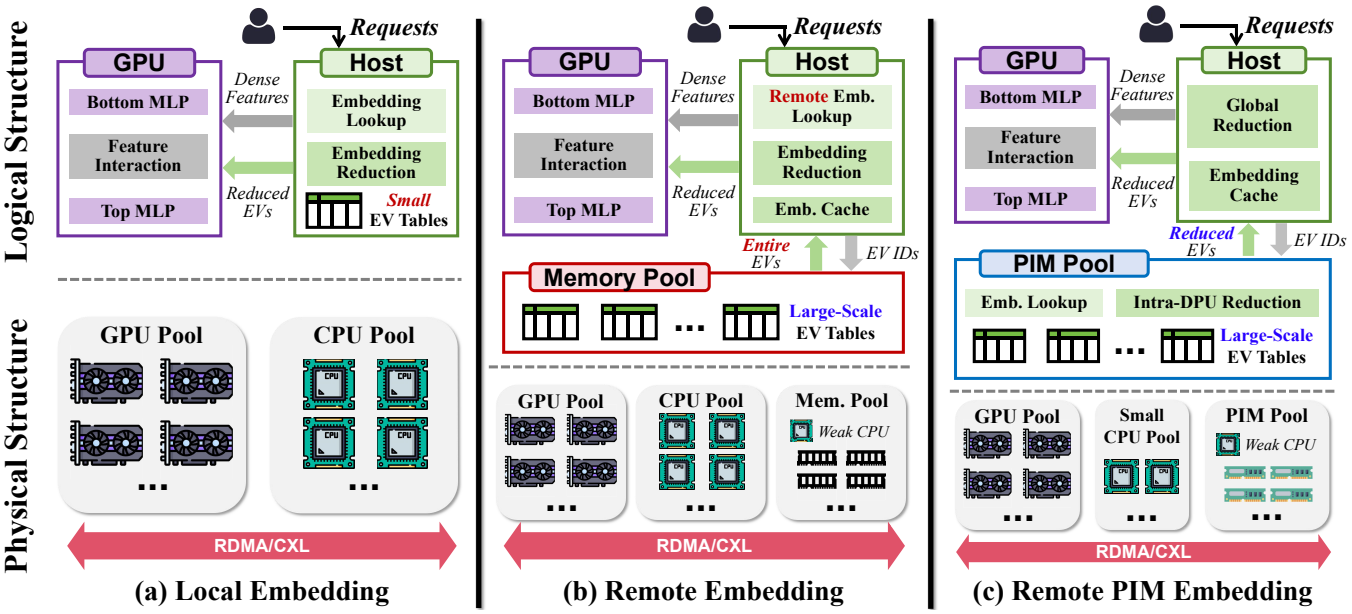


Figure 2: The illustration of different embedding disaggregation schemes *(the upper part shows the logical structures of different schemes, while the lower part displays the physical structures).*

age) and sparse features (e.g., user ID). Sparse features are transformed into trained d-dimensional embedding vectors (EVs) [3], stored in EV tables based on their properties. During inference, the system retrieves the EVs from tables, then reduces them using pooling mechanisms (e.g., sum/max). The dense features are also encoded into d-dimensional EVs via the bottom multi-layer perceptron (MLP). The combined EVs of dense and sparse features are then processed by the top MLP to output scores (e.g., click-through rates).

Despite diverse and ad-hoc designs [19,48,61,82], deep recommendation systems typically follow the *embedding-MLP paradigm* [24, 58], where an embedding layer encodes input features and a top MLP computes item scores. While recent approaches [4, 15, 77] incorporate large language models for recommendations, embedding-MLP-based DLRMs remain dominant due to their cost-effectiveness.

2.2 Embedding Disaggregation

In deep recommendation systems, embedding layers and MLP layers have distinct characteristics: MLP layers are compute-intensive and benefit from GPU processing, while embedding layers, with low compute intensity, are inefficient on GPUs. To optimize resource utilization and cost efficiency, these systems typically adopt an embedding-disaggregated structure [14, 58]. As shown in Figure 2(a), the embedding and MLP layers are respectively processed by CPUs and GPUs. Such a *local embedding* disaggregation allows flexible CPU/GPU configurations and high resource efficiency.

Modern deep recommendation systems store massive EVs to represent the features of billions of users and items, resulting in unprecedentedly large EV tables. For instance, an EV table can reach 2 terabytes when storing a billion 512-dimension EVs in FP32 format [44, 55], which exceeds the

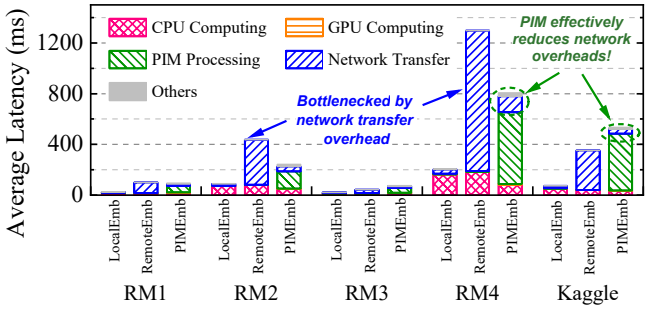


Figure 3: The latency breakdowns of different embedding disaggregation schemes.

memory capacity of a single machine. Memory disaggregation schemes [20, 26, 52, 53, 57, 63, 64, 78, 79] provide flexible and high-scalable memory management. By adopting memory disaggregation, deep recommendation systems accommodate large-scale EV tables while providing high performance and cost-efficiency [49, 67]. As shown in Figure 2(b), the *remote embedding* scheme stores the EV tables in remote memory pools. During inference, the host server retrieves *all* required EVs from the remote memory pool before locally performing embedding reduction. This design addresses the memory-capacity problem, but exposes every raw EV lookup to remote data movement. The remote access path can dominate inference latency even when the embedding tables are successfully disaggregated.

Unfortunately, despite using high-performance networks (e.g., RDMA [16] or CXL [11]) to connect these pools and caching hot EVs on the host server, remote embedding disaggregation still suffers from significant network overheads. As shown in Figure 3, data transfer overheads dominate the end-to-end latency, accounting for up to 80% across various

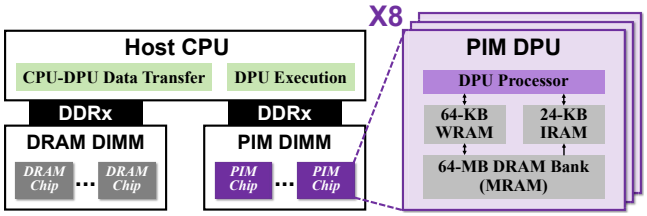
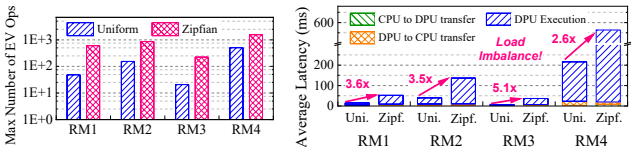


Figure 4: The architecture of a UPMEM-based PIM system.



(a) Maximum DPU Load (b) Latency Breakdown

Figure 5: The performance of PIM-enabled embedding disaggregation under the workloads of different skewness *(the number of EV operations indicates the number of EVs that need to be lookuped and reduced for the DPU).*

recommendation models (experimental details in §5.1). This is because recommendation systems must retrieve massive EVs during inference, ranging from tens to hundreds of EVs per table per request [23,56,82], thus leading to GB-scale data movement for each inference. This highlights the critical need for optimizing the I/O path in remote embedding schemes.

Processing-in-memory (PIM) [33, 34, 36–39, 42] is a computing paradigm that minimizes data movement by bringing computation closer to data (in or near the memory), offering high bandwidth and energy efficiency. PIM is a promising solution for reducing the network overhead of remote embedding disaggregation [18, 27, 40, 49]. As illustrated in Figure 2(c), by offloading embedding lookup and reduction tasks to the PIM pool, the *remote PIM embedding* scheme allows the host server to fetch only the *reduced* EVs, hence significantly reducing network transmission volume. Its key idea stems from a pragmatic design principle: introducing controlled, internal re-coupling within the disaggregated memory pool to eliminate the primary data transfer bottleneck, thereby enabling feasible and high-performance disaggregation at the system level. PIM pool acts as a specialized disaggregated resource, analogous to GPU pools, representing a natural evolution of disaggregated architectures toward heterogeneity and specialization.

2.3 Real PIM Systems for Embedding

While existing designs [18, 25, 32, 46, 71] leverage PIM to improve the application performance across diverse domains, they are mainly evaluated based on simulators (e.g., DRAMsim [47] or Ramulator [51]) or FPGA-based emulations [27]. UPMEM proposes commercial PIM hardware [13], boosting many applications with the real PIM system [6, 10, 30, 45].

2.3.1 The Architecture of A Real PIM System

Figure 4 illustrates the architecture of a real PIM system, UPMEM PIM, comprising a host CPU, standard DRAM modules, and PIM modules. The UPMEM PIM module is a DDR4-2400 double-rank DIMM with 16 PIM chips, each containing 8 DRAM processing units (DPUs). Each DPU includes a 64-MB main DRAM bank (called *MRAM*), a 64-KB scratchpad memory (*WRAM*), a 24-KB instruction memory (*IRAM*), and a DPU processor. The UPMEM PIM system achieves a high per-DIMM bandwidth of 80 GB/s and a total aggregated bandwidth of 1.6 TB/s [17].

The DPU processor offers high thread-level parallelism with 24 threads (*tasklets* in UPMEM [69]) that share MRAM/WRAM data and synchronize using primitives such as mutexes. However, each DPU is a lightweight processor with limited single-thread performance, rather than a latency-optimized CPU core. Its performance therefore depends on exposing enough independent work across tasklets and avoiding frequent synchronization. Furthermore, DPUs need to load data from MRAM into WRAM for processing. The DPU SDK facilitates programming by providing transparent MRAM access via a software cache. DPU programming adopts a conventional accelerator model, similar to CUDA [59]. The execution program (analogous to a kernel function) is typically preloaded onto DPUs. The host CPU transfers task data to DPUs, initiates the execution program, and retrieves results after the DPUs complete processing. The transmission sizes for DPUs involved in a single transfer must be *equal* to ensure data alignment and simplify hardware design [69]. This constraint results in a trade-off. A larger transfer group can efficiently utilize host-DPU bandwidth, but every DPU in the group must pad its buffer to the largest useful payload in that group.

2.3.2 Key Challenges in PIM-Based Embedding

Although PIM-enabled embedding disaggregation significantly reduces network costs, inefficient PIM processing becomes a new performance bottleneck, accounting for 63% of latency on average, as shown in Figure 3. This inefficiency arises from the following challenges:

Challenge 1: ***Inefficient intra-DPU parallelism.*** DPUs handle embedding lookups and reductions on *EV groups* (each group contains EVs from the same request and table). However, fully utilizing the parallelism of DPU threads to efficiently perform embedding reduction is challenging. A straightforward approach is to assign different EV groups to separate DPU threads, but this leads to load imbalance due to varying numbers of EVs in each group. A finer-grained strategy that assigns reduction tasks at the EV level eliminates load imbalance, but causes contentions among threads competing to reduce the same group. Consequently, this approach results in marginal performance improvements or even degrades performance due to locking overheads.

Challenge 2: ***Severe inter-DPU load imbalance.*** Although the PIM system offers high bandwidth and parallelism with thousands of DPUs, balancing workloads among these DPUs is critical. Each DPU stores a subset of EVs and performs

reduction on them. However, real-world workloads are often skewed, with certain hot EVs being accessed frequently [1, 29, 73, 75], resulting in significant inter-DPU load imbalance. While caching hot EVs at the host server can mitigate this imbalance, uncached EVs still vary in access frequency. Consequently, under skewed workloads (e.g., Zipfian distribution, $\alpha = 0.99$), some DPUs become overloaded, as shown in Figure 5(a). This load imbalance degrades PIM processing latency by up to 5.1×, as depicted in Figure 5(b).

Challenge 3: *Granularity mismatch in PIM processing.* While DPUs are processed at the *DPU-Set* granularity (including all allocated DPUs), task dispatch and execution are performed at the *single DPU* granularity. Each DPU exhibits different transfer and execution time due to the inter-DPU load imbalance, which is hard to eliminate in practice. Since DPUs must transfer data of the *same* size in a single transmission, this granularity mismatch forces less-loaded DPUs to pad and transfer extra padding data, leading to severe transfer amplification. Furthermore, during both data transfer and execution, all DPUs must wait for the slowest DPU, exacerbating the performance degradation.

3 Design Principles for Real PIM Devices

Achieving high-performance recommendation inference on disaggregated PIM systems needs to coalesce the properties of software and hardware. We propose three fundamental *system-level design principles* that address the core challenges of exploiting real PIM devices for data-intensive workloads such as recommendation inference. These principles capture unique hardware properties and provide a portable framework for designing high-performance PIM-enabled systems.

• **Principle 1: *Decouple data access via fine-grained partitioning*.** PIM units (e.g., DPUs) offer tens of hardware computation cores (e.g., DPU threads), but exploiting this parallelism is non-trivial due to load imbalance and synchronization overheads. A key insight is to decompose *monolithic* data vectors into *independent* dimension slices to enable concurrent, lock-free memory access within PIM units. For instance, in embedding reduction, by assigning each computation core a contiguous slice of dimensions for all vectors, we achieve efficient computation parallelism without locks or barriers. This principle is applicable to any data-parallel operation in which the computation can be partitioned along an independent axis (e.g., dimensions, columns, or keys). Such a decomposition maximizes core utilization while eliminating synchronization.

• **Principle 2: *Adaptive data placement for skewed workloads*.** PIM devices comprise thousands of independent PIM units. Static partitioning or replicating strategies fail to maintain load balance across PIM units under skewed, varying access patterns. An effective approach is to *dynamically redistribute hot data* across PIM units based on observed access frequency. This approach involves proactively replicating frequently accessed data items (e.g., EVs) onto multiple PIM units, enabling fast and efficient data redistribution. The system continuously adapts replica placement via lightweight split/merge operations, while controlling the trade-off between redundancy (for load balance) and locality (for reduction performance). This principle shifts from static data placement approaches to a data-centric, adaptive approach that maintains load balance under dynamic workloads.

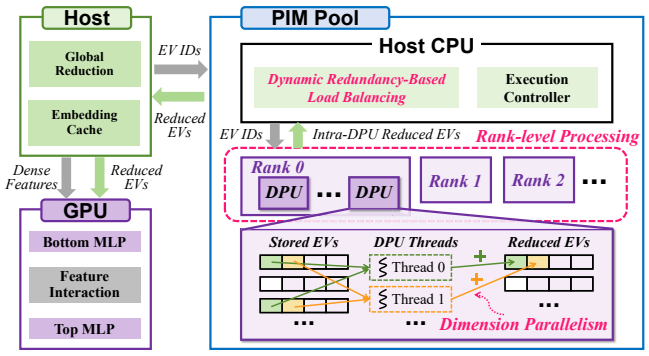


Figure 6: The system overview of CloverRec.

• **Principle 3: *Align host-device data transfer granularity with hardware hierarchy*.** The mismatch between the logical granularity of task dispatch and the physical granularity of data transfer causes significant overheads. To bridge this gap, the transfer granularity should align with the hardware's memory and interconnect hierarchy. By transferring data at the rank level, we minimize data padding overhead while saturating the available host-device bandwidth. This principle advocates for a granularity-aware data movement strategy that balances transfer amplification against bandwidth utilization, a critical trade-off in PIM-enabled system.

These design principles collectively address the key challenges of PIM programming: parallelism efficiency, dynamic load balancing, and data movement optimization. Although discussed in the context of recommendation inference and illustrated with UPMEM PIM as a case study in this work, these principles generalize to a broad class of data-intensive workloads and can be applied to diverse PIM devices and similar heterogeneous, near-memory architectures.

4 CloverRec Design

4.1 System Overview

Figure 6 shows the overview of the CloverRec system. CloverRec consists of three components, including (1) a *host server* that performs the global inter-DPU reduction, (2) a *GPU pool* that computes MLP layers, and (3) a *PIM pool* for storing large-scale EV tables and performing high-performance embedding operations (lookup and reduction). The PIM pool serves as a high-bandwidth, compute-capable storage tier, storing large-scale EV tables and performing data reduction to minimize data movements. It comprises PIM-enabled systems, each consisting of a host CPU, DRAM memory, and PIM-enabled memory.

Compared to conventional embedding disaggregation

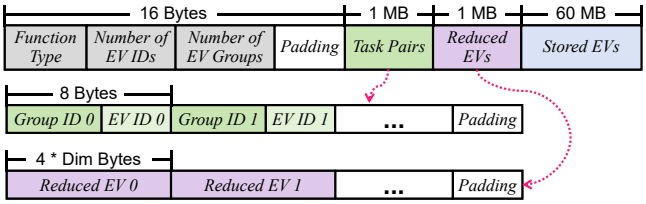


Figure 7: The structure of the DPU buffer.

schemes [14, 49, 58, 67], CloverRec's essential feature stems from the offloading of embedding operations to the PIM pool. During inference, the host server checks its embedding cache to obtain the cached EVs and sends the missed EV IDs to the PIM pool. The host CPU in a PIM system dispatches the EV IDs to DPUs and launches DPU execution to perform embedding operations. After the execution, the host CPU fetches the reduced EVs from DPUs and sends them to the host server. After receiving the reduced EVs, the host server performs the global reduction of these EVs and the cached EVs to generate the final output of embedding layer. Moreover, the host server fetches missed EVs to update the embedding cache. To minimize transfer cost, CloverRec only fetches a random portion of the missed EVs ($< 1\%$ of reduced EVs) to update the embedding cache.

Embedding tables are partitioned by rows and evenly distributed across all available DPUs, supporting arbitrary table sizes. Initially, DPUs store the same number of EVs. Each DPU maintains a buffer in MRAM, which stores EVs, embedding operations tasks, and reduction results. Figure 7 presents the detailed structure of the DPU buffer. Before execution, the host CPU sends the metadata (including the function type, the number of EV IDs, and the number of EV groups) and task pairs (including the group and EV IDs) to the DPU buffer. During execution, each DPU leverages multiple threads to concurrently perform embedding operations. Each EV group maintains a reduced EV as the reduction result. It is worth noting that each data transmission only involves the valid task pairs and reduced EVs while padding extra data to ensure the transfer sizes are the same.

While offloading embedding operations on real PIM systems poses several challenges (§2.3), CloverRec overcomes these challenges and achieves efficient PIM-enabled embedding operations with the following synergistic designs:

- ***Efficient intra-DPU parallelism (§4.2).*** CloverRec enables dimension-parallel embedding reduction across DPU threads, eliminating inter-thread synchronization while maximizing thread parallelism for high performance.
- ***Dynamic redundancy-based load balancing (§4.3).*** CloverRec balances inter-DPU loads by replicating hot EVs across multiple DPUs. It dynamically optimizes replication scheme through *split* and *merge* operations, adapting efficiently to varying access patterns.
- ***Rank-level PIM processing (§4.4).*** CloverRec employs rank-level transfer to simultaneously minimize transmission amplification and fully saturates the bandwidth between the host CPU and DPUs. By pipelining data transfers with DPU execution, CloverRec further reduces transfer overhead.

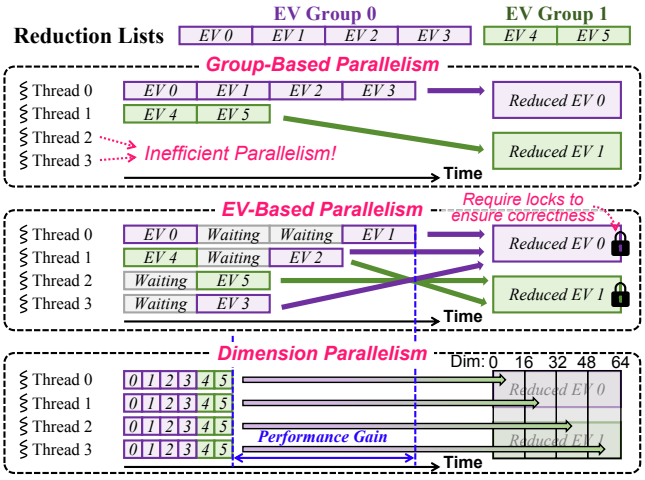


Figure 8: The illustration of different intra-DPU reduction parallelism mechanisms *(using 4 threads as an example)*.

4.2 Dimension Parallelism

DPU provides high parallelism with tens of DPU threads. It is crucial to fully leverage the parallelism of DPU threads for achieving high-performance embedding operations. Intuitively, the DPU allows DPU threads to perform embedding operations at the EV group granularity, called *group-based parallelism*. As shown in Figure 8, when processing 2 EV groups (containing 4 and 2 EVs respectively) with 4 threads, group-based parallelism leaves threads 2–3 idle, leading to low parallelism efficiency.

To improve utilization of DPU threads, DPUs can adopt finer-grained parallelism by arranging DPU threads to perform embedding operations at the EV granularity. This *EV-based parallelism* allows DPU threads to operate on individual EVs across groups, as shown in Figure 8. However, EV-based parallelism may assign different EVs from the same EV group to multiple DPU threads. Although EV lookups are read-only and conflict-free, the following reduction phase requires these threads to accumulate values into the same reduced EV buffer. Locks or atomic operations are therefore needed to prevent race conditions during concurrent updates, and the resulting synchronization overhead can dominate the lightweight embedding computation.

We observe that the EVs are typically high-dimensional to represent features of billion-scale data [19, 54, 58, 80], making it practical to follow *Design Principle 1 (§3)*. Based on this observation, CloverRec introduces *dimension parallelism* to fully leverage the parallelism of DPU threads. As shown in Figure 8, each DPU assigns threads to process embedding operations of distinct dimension ranges for all involved EVs. For instance, thread 0 processes the operations of dimensions 0–15 for EVs 0–5. The dimension parallelism maximizes thread utilization while eliminating inter-thread synchroniza-

tion, hence bringing significant performance gains.

There are two requirements for applying dimension parallelism. First, the number of EV dimensions needs to exceed the number of DPU threads. Second, the embedding operations should not involve inter-dimension dependencies and allow DPU threads to independently operate on distinct dimensions. These requirements are typically met in practice since: (1) enterprise-level recommendation models use high-dimensional EVs to improve accuracy [19, 54, 58, 80], and (2) common pooling operations (e.g., sum/max) are dimension-independent. Therefore, dimension parallelism can be effectively applied in a wide range of scenarios.

4.3 Dynamic Load Balancing

Since CloverRec leverages thousands of DPUs to achieve high-performance embedding operations, it is essential to ensure load balancing across DPUs. In practice, the load imbalance stems from the fact that the hotness of EVs exhibits significant differences. As a result, the DPUs that store hot EVs need to handle a large number of embedding operations and become overloaded. Prior designs [7, 10, 32] employ static load balancing mechanisms, incurring severe overheads when access pattern changes. In contrast, following *Design Principle 2 (§3)*, CloverRec leverages a dynamic load balancing scheme that adapts to varying access patterns with constrained overhead, providing finer-grained adjustment than static approaches.

4.3.1 Redundancy-based Load Balancing

CloverRec leverages a redundancy-based load-balancing mechanism to prevent DPUs from being overloaded. For a hot EV, CloverRec stores multiple replicas of the EV across multiple DPUs, each of which holds a replica. Each embedding operation on a hot EV is randomly dispatched to one of its replica-hosting DPUs.

In CloverRec, we quantify EV hotness using the number of accesses to an EV (i.e., EV frequency), a standard metric for identifying hotness in many systems such as caching systems [41, 74] and tiered-memory systems [43, 72, 83]. When enabling redundancy, each replica-hosting DPU has an equal probability of serving operations for that EV. The expected per-replica load (EV_Freq) is computed as $EV\_Freq = \frac{Total\_Freq}{Number\_of\_Replicas}$, where $Total\_Freq$ indicates the total access frequency of an EV. For non-redundant EVs, their numbers of replicas are 1. Each EV's total frequency is tracked via a 4-byte counter, adding negligible memory overheads ($< 1\%$ for 128-dimension EVs). For DPU_x storing K EVs, its expected frequency is computed as $DPU\_Freq_x = \sum_{i=0}^{K-1} EV\_Freq_i$, where EV_Freq_i is the expected frequency of the i^{th} EV (or replica) stored on DPU_x. The expected frequency of a DPU represents its estimated hotness during the following requests.

Figure 9 illustrates the effectiveness of redundancy-based load balancing. Consider 3 DPUs where DPU 1 initially stores a hot EV (EV 5) with frequency 850, causing overload.

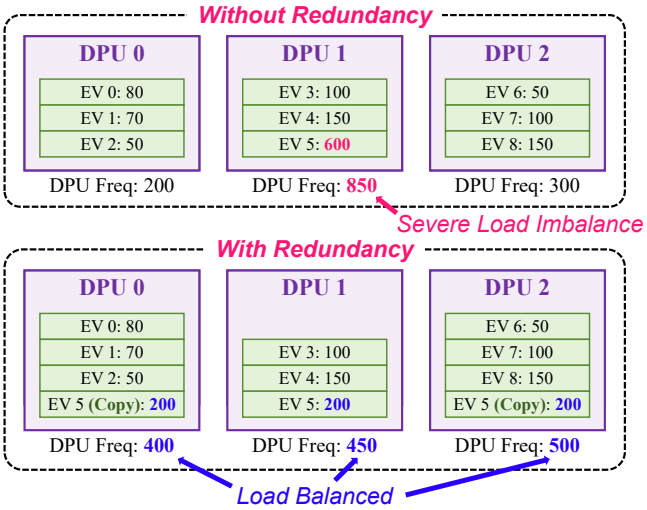


Figure 9: The illustration of redundancy-based load balancing *(using 3 DPUs as an example)*.

After replicating EV 5 to DPUs 0 and 2, the frequency of DPU 1 significantly decreases while the frequencies of DPUs 0 and 2 only slightly increase, balancing overall load. To minimize tracking overhead for DPU frequencies, we use a hybrid frequency method that consists of: (1) Increasing the actual DPU frequency when accessing an EV replica. (2) Updating expected DPU frequencies with the changes in expected EV frequencies when creating/deleting replicas. This hybrid approach maintains accuracy while minimizing the costs of updates.

To achieve adaptive load balancing across varying access patterns, CloverRec leverages a dynamic redundancy-based load balancing mechanism with four key design goals: (1) *Minimize the maximum DPU loads.* CloverRec replicates hot EVs onto cold DPUs to reduce the maximum DPU loads. (2) *Adaptive replica placement.* CloverRec leverages fine-grained *split* and *merge* operations to dynamically adjust the replica placement with low overheads. Moreover, CloverRec employs an aging mechanism to prevent overvaluing EVs. (3) *Balance between redundancy degree and EV locality.* It is a trade-off between a high redundancy degree (i.e., more replicas) and a high locality of EVs. CloverRec optimizes this trade-off by facilitating intra-DPU reduction and constraining the redundancy degree. (4) *Constrained runtime overhead.* To make the runtime overhead manageable, CloverRec dynamically tunes the operations rates of split/merge operations based on real-time overhead measurements.

4.3.2 Split Operation

CloverRec creates a replica with a *split* operation, which consists of four steps. (1) Hot DPU selection, which chooses the DPU with highest frequency. (2) Hot EV selection, which samples EVs from the selected hot DPU to efficiently identify the one brings the highest reduction in DPU frequency, i.e., the expected frequency of the EV would decrease the most after creating a new replica. (3) Cold DPU selection, instead of

simply selecting DPUs with the lowest expected frequencies, it prioritizes to select DPUs that (a) contain EVs from fewer distinct tables, and (b) contain more EVs from the same table as selected hot EV. Since EVs from the same table are more likely to be involved in the same EV group, this selection approach facilitates intra-DPU reductions. Moreover, to ensure that the split operation reduces the maximum of the expected DPU frequency, there is a constraint for the selection: the frequency of the cold DPU after storing the new replica does not exceed the frequency of the selected hot DPU. The DPUs with insufficient memory for the new replica are skipped in this step. (4) Replica placement, which creates a new replica of the selected hot EV on the selected cold DPU.

4.3.3 Merge Operation

CloverRec maintains a maximum number of total replicas ($MAXN_{rep}$) to constrain the redundancy degree for limiting CPU computation costs, EV transfer overheads, and extra memory footprints. When the current number of total replicas approaches the maximum number (e.g., exceeding 95% of the maximum number), CloverRec deletes the replicas with *merge* operations, each of which consists of three steps: (1) Cold EV selection, which samples EVs to identify the one whose expected frequency would increase least upon replica deletion. (2) Hot DPU selection, which chooses the DPU with the highest frequency among DPUs storing the selected EV's replicas. (3) Replica deletion, which removes the replica from the selected DPU.

4.3.4 Aging Mechanism

Since CloverRec uses frequency-based hotness identification, the hot EVs can be overvalued when they become cold. To mitigate this problem, CloverRec employs an aging mechanism, which is a cost-efficient and widely-used way to capture object recency by simply increasing the gained scores of recently accessed EVs [9]. When an EV is accessed, its total frequency increases by $Freq_{gain}$, which is initialized to 1. CloverRec increases the $Freq_{gain}$ by 0.01 every N_{aging} requests. The N_{aging} is a user-configured hyperparameter. With the aid of aging, the recently hot EVs can be efficiently identified. We recommend setting N_{aging} at 10–100 according to experimental results in §5.5.

4.3.5 Adaptive Parameter Changing

CloverRec dynamically tunes load balancing based on real-time overhead measurements, including two key mechanisms. (1) Operation rate control, which adjusts split/merge operations per inference (i.e., N_{split} and N_{merge}). When the split overhead is low (e.g., $< 1\%$ of the embedding operation latency) and more replicas are needed (e.g., the current total replicas are less than 95% of the maximum), CloverRec increases N_{split}, while decreasing it when the overhead is high (e.g., $> 3\%$ of the embedding operation latency). CloverRec applies a similar adaptive approach to adjust N_{merge}. (2) Replica capacity (i.e., $MAXN_{rep}$) management, which freezes replica creation if the CPU computation and EV transfer overhead of increasing the redundancy degree overwhelms the performance benefits. CloverRec halts the creation of new replicas by setting the $MAXN_{rep}$ to the current number of total replicas. If the performance improvements from redundancy outweigh the additional costs, CloverRec gradually resumes creating replicas by increasing the $MAXN_{rep}$. We recommend initializing $MAXN_{rep}$ at 1–5% of total unique vectors. This ensures significant load balancing headroom while maintaining low memory overhead.

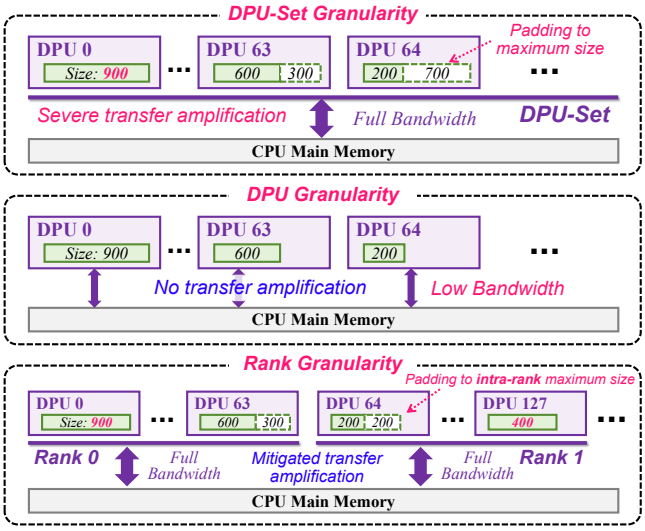


Figure 10: The illustration of different transfer granularities.

4.4 Rank-Level PIM Processing

The host CPU typically manages PIM processing at the *DPU-Set* granularity [69], which transfers data to all allocated DPUs with a single copy function (e.g., `dpu_push_xfer` and `dpu_push_sg_xfer`) and launches DPUs with a single execution function (e.g., `dpu_launch`). Different DPUs typically have different transfer loads because sparse-feature accesses are skewed and each request may generate different numbers of task pairs and reduced EVs across DPUs. However, the transfer sizes for the involved DPUs need to be the same to ensure data alignment. As a result, DPUs with fewer transfer loads need to pad the transmission buffer to the maximum transfer size, leading to severe transfer amplification. For instance, in Figure 10, consider DPU_0 and DPU_{64} need to transfer 900 and 200 reduced EVs to the host CPU respectively, the DPU_{64} needs to pad its transfer load by 4.5×. While the DPU-Set-level processing can saturate the bandwidth between the host and DPUs, the transfer amplification significantly degrades the transfer efficiency.

An intuitive way to eliminate the transfer amplification is to carry out the PIM processing at the *single DPU* granularity. However, this approach suffers from severe bandwidth underutilization. As shown in prior study [17], DPU-level transfers achieve 0.27 GB/s (CPU-to-DPU) and 0.12 GB/s (DPU-to-CPU) sustained bandwidth, which are only 4.0% and 2.5% of the full bandwidths 6.68 GB/s (CPU-to-DPU)

and 4.74 GB/s (DPU-to-CPU) respectively. This bandwidth bottleneck ultimately degrades overall performance.

CloverRec optimizes this granularity trade-off through rank-level processing. This design is the concrete practice of *Design Principle 3 (§3)* on UPMEM PIM. CloverRec partitions DPUs according to the physical rank organization of UPMEM DIMMs and issues one copy operation for each rank. For a rank, CloverRec pads each DPU buffer to the maximum useful payload among DPUs in this rank, instead of the maximum payload among all allocated DPUs. This rank-level transfer mitigates transfer amplification while still saturating the bandwidths between CPU and DPUs [17]. CloverRec further pipelines data transfer with DPU execution through asynchronous copy and execution calls, which alleviates the transfer imbalance and improves the performance.

5 Performance Evaluation

5.1 Experimental Setup

We incorporate CloverRec into the representative open-source recommendation framework (DLRM) [14, 58] and evaluate it on a ***real disaggregated PIM system***.

Platform. We conduct experiments with three servers (a GPU server, a PIM-enabled server, and a host server), each with a 100 Gbps Mellanox ConnectX-5 InfiniBand RNIC. These servers are interconnected through a 100 Gbps Mellanox SB7890 InfiniBand switch. We use the mature RDMA technology for implementations because the current UPMEM PIM product does not support CXL technology. The GPU server contains 1 NVIDIA Tesla V100 GPU, which handles the MLP layers. The PIM-enabled server is equipped with 8 UPMEM DIMMs and 192 GB DRAM, serving as the memory pool (in the remote embedding scheme) and the PIM pool (in the PIM-based embedding scheme), respectively. The 8 PIM DIMMs contain 1020 DPUs running at 350 MHz while 4 DPUs are non-functional and cannot be allocated[1]. The host server is employed to manage recommendation inference workflows and coordinate with the GPU server and the memory/PIM pool, equipped with two Intel 26-core Xeon Gold 6230R CPUs and 192 GB DRAM.

Comparisons. We compare CloverRec with state-of-the-art embedding disaggregation schemes (§2.2): local embedding (***LocalEmb***) [14], remote embedding (***RemoteEmb***) [49], and remote PIM embedding (***PIMEmb***). While LocalEmb does not support large-scale EVs and cannot be adopted in real-world recommendation systems, we evaluate it to demonstrate the performance upper bound. Besides, we evaluate PIMEmb as it provides a clean baseline to isolate and quantify the individual contribution of our techniques on the same hardware platform. For RemoteEmb, the host server uses one-sided RDMA read operations to fetch EVs from the remote memory pool.

Workloads. We evaluate CloverRec and the compared schemes using both synthetic and real-world workloads.

[1]This is a common case and has also been observed in prior works [6, 17].

Table 1: The different configurations of DLRM.

Model	Number of EV Tables	Number of Rows/Table	Emb. Dim	Number of Lookups/Table	Bottom-MLP	Top-MLP
RM1 [22]	10	1 Million	64	80	256-128-64	256-64-1
RM2 [22]	40	1 Million	64	80	256-128-64	512-128-1
RM3 [22]	10	1 Million	64	20	2560-512-64	512-128-1
RM4 [49]	80	1 Million	128	160	512-256-128	512-128-1
Kaggle [29]	26	Variable (up to 10 M)	64	80	13-512-256-64	512-256-1

Specifically, the synthetic workloads are generated by using 4 different configurations of DLRM (RM1–4), as shown in Table 1. These workloads are generated with a Zipfian distribution ($\alpha = 0.99$ by default). Among these workloads, RM1, RM2, and RM4 involve large EV tables and heavy embedding operations, making them embedding-intensive. On the other hand, RM3 is more compute-intensive with larger MLP layers. We further use open-source real-world workload Criteo-Kaggle [29], which provides features and click feedback data for millions of advertisements. In this workload, consecutive records are grouped to form a request.

Default configurations. Unless otherwise stated, we perform recommendation inference in a batched manner with a batch size of 32. For PIMEmb and CloverRec, we use all 1020 DPUs to ensure that the EV tables can be fully stored in DPUs and use one CPU core in the remote PIM pool to initialize the network connection and manage the workflow of PIM-enabled embedding operations. For each DPU, we launch 16 DPU threads to execute intra-DPU embedding operations. For LocalEmb and RemoteEmb, we use 52 CPU threads to perform embedding operations. For RemoteEmb, we use one CPU core to initialize the network connection of the remote memory pool. For PIMEmb, we adopt group-based parallelism to perform intra-DPU reductions while transferring data between DPUs and the host CPU at the DPU-Set granularity. Moreover, PIMEmb does not employ a load-balancing mechanism. For RemoteEmb, PIMEmb, and CloverRec, the embedding cache adopts LRU caching scheme [12]. The cache size is set to 0.1% of the total size of all EVs by default, since the memory capacity of the host server and the memory pool is several orders of magnitude different. Besides, we set the aging parameter N_{aging} of CloverRec to 100 by default.

5.2 End-to-End Performance

Figure 11 presents the throughput-latency curves of different embedding disaggregation schemes. To plot a throughput-latency curve, we record the throughput and latency via various batch sizes. At small batch sizes, low request concurrency may underutilize DPU parallelism, while fixed host-DPU coordination overheads remain. As the batch size increases, DPU parallelism becomes better utilized and the benefit of reduction offloading dominates these fixed overheads. Therefore, CloverRec shows more gains in the high-throughput region, which matches the high-concurrency scenario targeted by production recommendation inference.

Embedding-intensive workloads *(RM1, RM2, RM4, and*

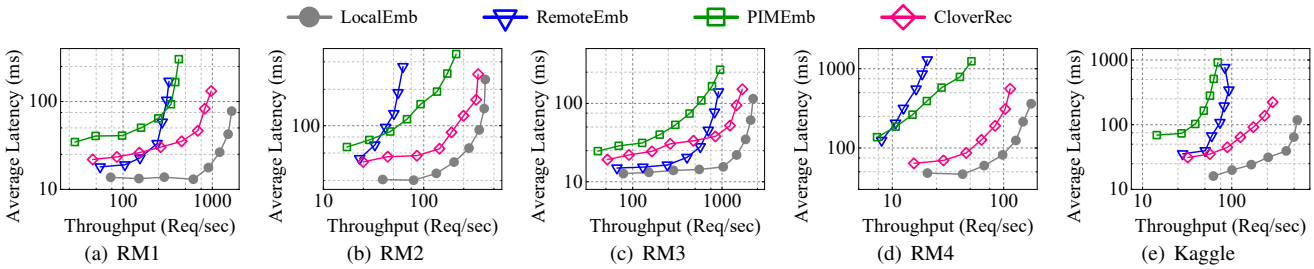


Figure 11: The throughputs and latencies of different schemes under various workloads.

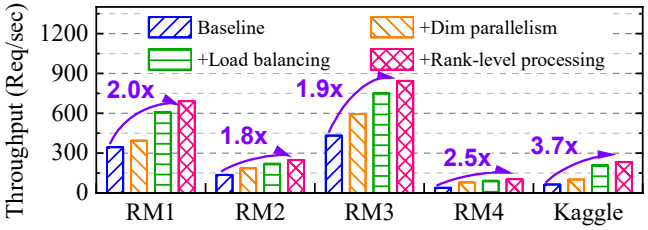


Figure 12: The ablation study of CloverRec designs.

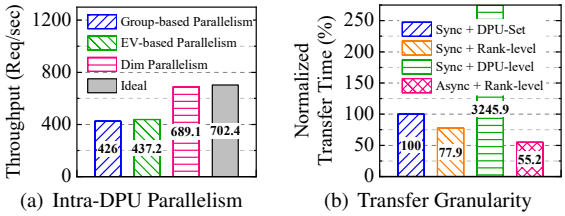


Figure 13: The performance of different intra-DPU parallelism and transfer granularities *(enabling other designs)*.

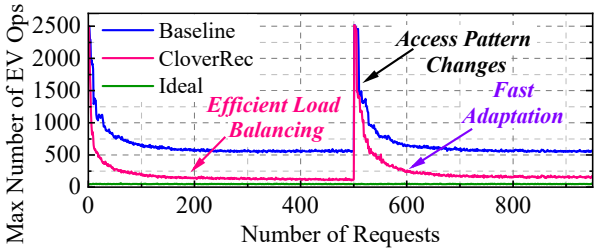


Figure 14: The maximum DPU loads of different schemes with changes of access patterns.

Kaggle). The results indicate that RemoteEmb suffers from low throughput due to significant network transfer overhead. Compared with RemoteEmb, CloverRec achieves up to an 8.18× improvement in throughput. The improvements is primarily attributed to the reduction in effective data movement volume and the elimination of network-bound stalls. Moreover, compared with LocalEmb, which is considered the performance upper bound, CloverRec provides comparable performance, highlighting the effectiveness and significance of PIM-enabled operation offloading. In contrast, PIMEmb provides limited or even worse performance compared with RemoteEmb due to inefficient PIM processing and load imbalance. The load imbalance problem becomes worse with larger batches, which further exacerbates the inference performance. By ensuring efficient intra- and inter-DPU processing and balanced load distribution, CloverRec achieves throughput improvements of up to 5.05× over PIMEmb.

Compute-intensive workload *(RM3).* Although the embedding layer in RM3 contributes less to overall runtime compared with embedding-intensive workloads, embedding operations still dominate inference performance. CloverRec outperforms RemoteEmb and PIMEmb by up to 1.49× and 2.09×, respectively, demonstrating its efficiency and effectiveness across diverse workloads.

5.3 Ablation Study

To understand the benefits of CloverRec designs, we evaluate their performance contributions through an ablation study, shown in Figure 12. Starting from a *Baseline* without any proposed designs, we progressively apply each feature.

+ Dimension parallelism. Dimension parallelism boosts performance by 1.14× (RM1) to 1.93× (RM4), demonstrating its importance for embedding-intensive workloads, where intra-DPU embedding operations significantly impact end-to-end inference efficiency. Figure 13(a) compares different parallelism mechanisms on the RM1 workload. While EV-based parallelism achieves load balancing across DPU threads, correctness guarantee mechanisms such as locking, resulting in severe inter-thread synchronization overheads and limited performance improvements. In contrast, dimension parallelism achieves thread load balancing without synchronization overhead, delivering 1.62× higher performance.

+ Dynamic redundancy-based load balancing. Achieving load balancing across DPUs improves throughput by 1.45× on average across all workloads, as CloverRec efficiently utilizes the high parallelism of thousands of DPUs with balanced loads. To evaluate the adaptability of its dynamic redundancy-based load-balancing mechanism, we shift the hotness EV IDs during RM1 execution, as illustrated in Figure 14, representing a change in access patterns. The *Ideal* denotes PIMEmb performance under uniform distribution. While access pattern changes increase maximum DPU load, CloverRec mitigates load imbalance by replicating hot EVs across multiple DPUs, reducing the maximum DPU load by 4.28× and significantly enhancing end-to-end performance. Furthermore, CloverRec achieves comparable maximum DPU load to *Ideal*, highlighting the effectiveness of its redundancy-based load balancing.

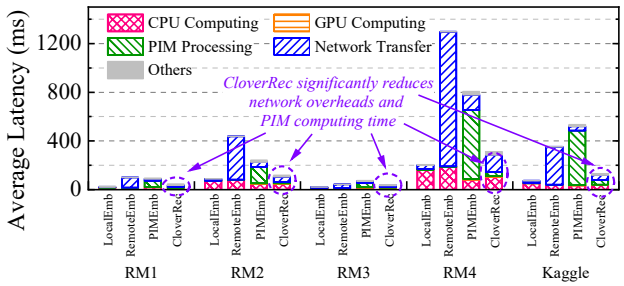


Figure 15: The latency breakdowns of different embedding disaggregation schemes.

+ Rank-level PIM processing. Rank-level PIM processing improves performance by 1.11× (RM2) to 1.13× (RM4) by reducing transfer amplification and maximizing bandwidth utilization between DPUs and the host CPU. Figure 13(b) shows the transfer time of CloverRec with different PIM processing mechanisms on RM4. Fine-grained transfer eliminates transfer amplification using DPU transfer granularity but suffers from low bandwidth, hence increasing transfer time. In contrast, rank-level transfer balances low amplification and high bandwidth, reducing transfer time by 22.1% compared to DPU-Set transfer granularity. CloverRec further pipelines data transfer and DPU execution, achieving an additional 22.7% reduction in transfer time.

5.4 Sources of Improvements

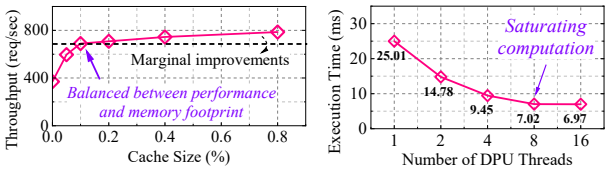
We analyze the sources of performance improvements in CloverRec by presenting latency breakdowns of different schemes, as shown in Figure 15. By leveraging efficient PIM-enabled remote embedding operations, CloverRec reduces network costs by up to 7.92× compared with RemoteEmb. Moreover, CloverRec enhances PIM-enabled embedding processing by up to 18.68× over PIMEmb through optimized intra-DPU parallelism, inter-DPU load balancing, and rank-level PIM processing. By proactively managing redundancy degrees, CloverRec minimizes PIM computing time while avoiding excessive computation and transfer overheads.

5.5 Sensitivity Analysis

We analyze the impact of the host cache size, the number of DPU threads, workload skewness, and the aging parameter on CloverRec's performance. We show the results on the typical workload RM1 due to space limitations, as similar trends are observed across other workloads.

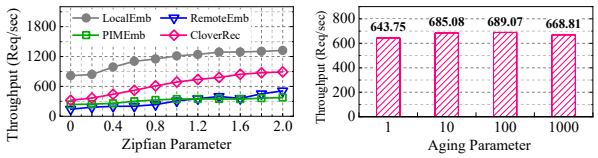
The host cache size. Figure 16(a) illustrates the performance of CloverRec under varying different host cache sizes. Increasing the cache size from 0 to 0.1% boosts throughput by 1.85×, highlighting the effectiveness of the host-side embedding cache. However, further increasing the cache size to 0.8% results in only marginal gains of 1.14×. Hence, CloverRec sets the default cache size at 0.1% to balance performance and memory overhead.

The number of DPU threads. Figure 16(b) shows the DPU execution time of CloverRec under varying numbers of



(a) The host cache size (b) The number of DPU Threads

Figure 16: The performance of CloverRec using different host cache sizes and numbers of DPU threads.



(a) Skewness (b) Aging

Figure 17: The throughputs of different schemes under various configurations.

DPU threads. Increasing DPU threads from 1 to 8 reduces execution time by 3.56×, exhibiting a sublinear scaling trend. Beyond 8 threads, performance gains become negligible, as 16 threads fully utilize the DPU processor pipeline and saturate computation resources. These findings align with prior research [17], which reports that 11 threads are sufficient to saturate computation resources for all arithmetic operations.

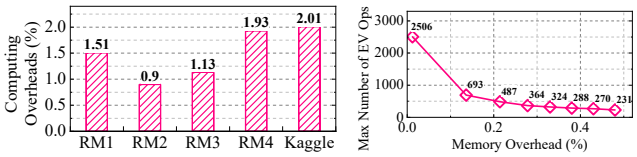
Skewness of workloads. We evaluate the impact of workload skewness on performance using various Zipfian parameters, as shown in Figure 17(a). Higher Zipfian parameters indicate more skewed workloads. For LocalEmb, higher skewness improves EV time locality, resulting in more cache hits on the CPU and enhancing performance. For RemoteEmb, PIMEmb, and CloverRec, higher skewness boosts throughput as embedding caches absorb more EV accesses, reducing transfer overhead. Compared with PIMEmb, CloverRec achieves better performance under the same skewness by efficiently identifying hot EVs and creating replicas for them.

Aging parameter. Figure 17(b) shows the effect of the aging parameter on CloverRec's performance. A large aging parameter prioritizes EVs with high frequencies, potentially leading to overvaluation and replication of cold EVs. On the other hand, a small aging parameter prioritizes recently accessed EVs. Overall, CloverRec demonstrates robustness to aging parameter variations, with only a slight performance fluctuation of 7.1%.

5.6 Overheads of CloverRec

Computing overheads. CloverRec introduces negligible computing overheads for split and merge operations to achieve load balancing. Figure 18(a) shows these overheads, including inter-DPU data transfer during split operations, ranging from 0.82% to 2.01% across workloads. CloverRec proactively constrains overheads by dynamically adjusting N_{split} and N_{merge} for split and merge operations.

Memory overheads. Figure 18(b) shows the maximum DPU load under varying memory overheads (the host cache is



(a) Computing Overheads (b) Memory Overheads

Figure 18: The overheads of CloverRec.

disabled to avoid its influence). Increasing replicas reduces maximum DPU load but slightly increases memory overhead. CloverRec achieves effective load balancing with only a few replicas, introducing trivial memory overheads within 0.5%.

6 Discussions

Beyond UPMEM PIM. We leverage the UPMEM PIM as a case study to demonstrate the efficiency and efficacy of CloverRec, since other PIM devices [36, 39] are not commercially available. Although PIM devices differ in memory organization, programming interfaces, and command sets, CloverRec's design principles are independent of UPMEM-specific mechanisms. We implement these principles on UPMEM PIM as a case study, while the core ideas, including dimension-split computation, adaptive replication, and granularity-aware data movement, target common properties of PIM-like systems, such as abundant lightweight parallel units, skewed memory-side workloads, and host-device transfer bottlenecks. Thus, CloverRec generalizes at the design-principle level, while each platform can instantiate the transfer granularity according to its own bandwidth hierarchy and alignment constraints.

The generality of the PIM pool for future connection technologies. While emerging connection technologies, like CXL [11], reduce network costs, memory-disaggregated embedding schemes still have to handle network bottlenecks due to heavy data transfer loads. Moreover, despite offering higher performance than RDMA, CXL's bandwidth (128 GB/s) remains 12.8× lower than that of PIM products (1.6 TB/s). Future advancements in PIM technology will further widen this performance gap. Therefore, offloading operations to the PIM pool remains a cost-efficient and promising solution for memory-disaggregated embedding schemes, even with future connection technologies.

7 Related Work

PIM hardware architectures. PIM has long been studied to mitigate the *"Memory-Wall"* problem [31, 66], but practical adoption was limited by memory technologies. Recent hardware prototypes and products have emerged [13, 33, 34, 36–39, 42], bringing PIM closer to practical adoption. These include Samsung's HBM-based PIM system [39], SK Hynix GDDR6-based PIM [36, 38], and AxDIMM's FPGA-enhanced DIMMs [33, 34, 42]. Unlike more specialized designs, UPMEM [13] integrates general-purpose compute cores into traditional 2D DRAM arrays, which CloverRec leverages to build a real disaggregated PIM system.

Applications using commercial PIM devices. Prior work has explored commercial UPMEM PIM for databases and storage [5, 10, 30], transactional memory [50], deep learning [45], and graph processing [6]. CloverRec instead presents a cost-efficient disaggregated recommendation system by offloading embedding operations to a PIM pool. The closest work, UpDLRM [7], targets monolithic recommendation scenarios and relies on EV-based parallelism, static partitioning, and data transfer, which limits adaptability to dynamic access patterns and causes suboptimal bandwidth utilization and amplified data movement. In contrast, CloverRec combines dimension parallelism, dynamic load balancing, and rank-level processing for scalable, resource-efficient, and high-performance disaggregated recommendation inference. Beyond recommendation, CloverRec's data-centric design can benefit other memory-bound workloads, such as LLM decoding [81], graph analytics [8], and vector search [68].

Simulation- and emulation-based PIM recommendation systems. Prior works [32, 40, 49] leverage hardware simulation and emulation to explore PIM-based solutions for recommendation systems. TensorDIMM [40] emulates PIM nodes using NVIDIA GPUs. RecNMP [32] simulates its PIM-related designs using a DRAM simulator Ramulator [51]. ReCXL [49] simulates PIM-enabled CXL device with modified DRAM simulators. However, these simulation- and emulation-driven designs require hardware modifications, limiting practical deployment. In contrast, our CloverRec demonstrates end-to-end performance on ***a real disaggregated PIM system***, coalescing UPMEM PIM DIMMs, NVIDIA GPUs, and Mellanox InfiniBand RNICs. To the best of our knowledge, our CloverRec is the first real system implementation of disaggregated PIM system, providing system-level insights on PIM hardware and significantly outperforming state-of-the-art embedding disaggregation schemes.

8 Conclusion

In this paper, we present, implement, and evaluate CloverRec, a cost-efficient recommendation inference system built on a real PIM product. CloverRec offloads bandwidth-intensive embedding operations to a remote PIM pool to reduce network costs. CloverRec adopts dimension parallelism to maximize the utilization of all DPU threads while minimizing inter-thread synchronization overhead. CloverRec further leverages a dynamic redundancy-based mechanism to achieve load balancing across DPUs. Moreover, CloverRec employs rank-level PIM processing to mitigate transfer amplification while saturating the bandwidth between the host CPU and DPUs. Our evaluation shows that CloverRec outperforms state-of-the-art embedding disaggregation schemes by several times.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022. We are grateful to our shepherd, Hui Lu, and anonymous reviewers for their comments and suggestions.

References

[1] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, pages 348–363. 2023

[2] Ehsan K. Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Jens Axboe, Valmiki Rampersad, Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, Hector Yuen, Dheevatsa Mudigere, Shishir Juluri, Akshat Nanda, Manoj Wodekar, Krishnakumar Nair, Maxim Naumov, Chris Petersen, Mikhail Smelyanskiy, and Vijay Rao. Supporting massive DLRM inference through software defined memory. In *Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems (ICDCS'22)*, pages 302–312. 2022

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*. 2015

[4] Keqin Bao, Jizhi Zhang, Yang Zhang, Wenjie Wang, Fuli Feng, and Xiangnan He. Tallrec: An effective and efficient tuning framework to align large language model with recommendation. In *Proceedings of the 17th ACM Conference on Recommender Systems (RecSys'23)*, pages 1007–1014. 2023

[5] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. pimdb: From main-memory DBMS to processing-in-memory dbms-engines on intelligent memories. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN'23)*, pages 44–52. 2023. https://doi.org/10.1145/3592980.3595312

[6] Shuangyu Cai, Boyu Tian, Huanchen Zhang, and Mingyu Gao. Pimpam: Efficient graph pattern matching on real processing-in-memory hardware. *Proceedings of the ACM on Management of Data (SIGMOD)*, 2(3):161, 2024

[7] Sitian Chen, Haobin Tan, Amelie Chi Zhou, Yusen Li, and Pavan Balaji. Updlrm: Accelerating personalized recommendation using real-world PIM architecture. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC'24)*, pages 211:1–211:6. 2024

[8] Weijian Chen, Shuibing He, Haoyang Qu, and Xuechen Zhang. Leapgnn: Accelerating distributed GNN training leveraging feature-centric model migration. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, pages 255–270. 2025

[9] Audrey Cheng, David Chu, Terrance Li, Jason Chan, Natacha Crooks, Joseph M. Hellerstein, Ion Stoica, and Xiangyao Yu. Take out the trache: Maximizing (tra)nsactional ca(che) hit rate. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. 2023

[10] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. Pimlex: A high-performance learned index with processing-in-memory. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST'25)*, pages 287–303. 2025

[11] CXL. Compute Express Link Specification. https://computeexpresslink.org/, 2025.

[12] Asit Dan and Donald F. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems (SIGMETRICS'90)*, pages 143–152. 1990

[13] Fabrice Devaux. The true processing in memory accelerator. In *Proceedings of the 31st IEEE Hot Chips Symposium (HCS'19)*, pages 1–24. 2019

[14] Facebook. Deep Learning Recommendation Model for Personalization and Recommendation Systems. https://github.com/facebookresearch/dlrm, 2019.

[15] Luke Friedman, Sameer Ahuja, David Allen, Zhenning Tan, Hakim Sidahmed, Changbo Long, Jun Xie, Gabriel Schubiner, Ajay Patel, Harsh Lara, Brian Chu, Zexi Chen, and Manoj Tiwari. Leveraging large language models in conversational recommender systems. *CoRR*, abs/2305.07961, 2023

[16] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 249–264. 2016

[17] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *CoRR*, abs/2105.03814, 2021

[18] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi R. Iyer, and Reetuparna Das. PIM is all you need: A cxl-enabled gpu-free system for large language model inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*, pages 862–881. ACM, 2025

[19] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: A factorization-machine based neural network for CTR prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI'17)*, pages 1725–1731. 2017

[20] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: a hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, pages 417–433. 2022

[21] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. WTF: the who to follow service at twitter. In *Proceedings of the 22nd International World Wide Web Conference (WWW'13)*. 2013

[22] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'20)*, pages 982–995. 2020

[23] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of facebook's dnn-based personalized recommendation. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*, pages 488–501. 2020

[24] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web (WWW'17)*, pages 173–182. 2017

[25] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. Neupims: NPU-PIM heterogeneous acceleration for batched LLM inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, pages 722–737. 2024

[26] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. Aceso: Achieving efficient fault tolerance in memory-disaggregated key-value stores. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP'24)*, pages 127–143. 2024

[27] Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefler, and Gustavo Alonso. Chameleon: a heterogeneous and disaggregated accelerator system for retrieval-augmented language models. *Proceedings of the VLDB Endowment (VLDB)*, 18(1):42–52, 2024

[28] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter C. Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David A. Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'21)*. 2021

[29] Kaggle. Display Advertising Challenge. https://www.kaggle.com/c/criteo-display-ad-challenge, 2014.

[30] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. Pim-tree: A skew-resistant index for processing-in-memory. *Proceedings of the VLDB Endowment (VLDB)*, 16(4):946–958, 2022

[31] William H. Kautz. Cellular logic-in-memory arrays. *IEEE Transactions on Computers*, 18(8):719–727, 1969. https://doi.org/10.1109/T-C.1969.222754

[32] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'20)*, pages 790–803. 2020

[33] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shinhaeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, IlKwon Yun, Sung Joo Park, Hyunsun Park, Joon-Ho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. Near-memory processing in action: Accelerating personalized recommendation with axdimm. *IEEE Micro*, 42(1):116–127, 2022

[34] Jin Hyun Kim, Shinhaeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, Joon-Ho Song, Jeonghyeon Cho, Kyomin Sohn, and Nam Sung Kim. Aquabolt-xl hbm2-pim, LPDDR5-PIM with in-memory processing, and AXDIMM with acceleration buffer. *IEEE Micro*, 42(3):20–30, 2022

[35] Daniar Heri Kurniawan, Ruipu Wang, Kahfi S. Zulkifli, Fandi A. Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. Evstore: Storage and caching capabilities for scaling embedding tables in deep recommendation systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 281–294. 2023

[36] Dae-Han Kwon, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyu-Dong Hwang, Jeongje Park, Kyeong Pil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, Jaewook Lee, Donguc Ko, Younggun Jun, Ilwoong Kim, Choungki Song, Ilkon Kim, Chanwook Park, Seho Kim, Chunseok Jeong, Euicheol Lim, Dongkyun Kim, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. A 1ynm 1.25v 8gb 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops MAC operation and various activation functions for deep learning application. *IEEE Journal of Solid-State Circuits*, 58(1):291–302, 2023

[37] Yongkee Kwon, Guhyun Kim, Nahsung Kim, Woojae Shin, Jongsoon Won, Hyunha Joo, Haerang Choi, Byeongju An, Gyeongcheol Shin, Dayeon Yun, Jeongbin Kim, Changhyun Kim, Ilkon Kim, Jaehan Park, Chanwook Park, Yosub Song, Byeongsu Yang, Hyeongdeok Lee, Seungyeong Park, Wonjun Lee, Seongju Lee, Kyuyoung Kim, Daehan Kwon, Chunseok Jeong, John Kim, Euicheol Lim, and Junhyun Chun. Memory-centric computing with SK hynix's domain-specific memory. In *Proceedings of the 35th IEEE Hot Chips Symposium (HCS'23)*, pages 1–26. 2023

[38] Yongkee Kwon, Kornijcuk Vladimir, Nahsung Kim, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, Jeongbin Kim, Jaewook Lee, Ilkon Kim, Jaehan Park, Chanwook Park, Yosub Song, Byeongsu Yang, Hyungdeok Lee, Seho Kim, Daehan Kwon, Seong Ju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeong Pil Kang, Jungyeon Kim, Junyeol Jeon, Myeongjun Lee, Minyoung Shin, Minhwan Shin, Jaekyung Cha, Changson Jung, Kijoon Chang, Chunseok Jeong, Euicheol Lim, Il Park, and Junhyun Chun. System architecture and software stack for gddr6-aim. In *Proceedings of the 34th IEEE Hot Chips Symposium (HCS'22)*, pages 1–25. 2022

[39] Young-Cheon Kwon, Sukhan Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je-Min Ryu, Jong-Pil Son, Seongil O, Hak-soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyunsung Shin, Jin Kim, BengSeng Phuah, Hyoungmin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, Sooyoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, Joon-Ho Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 25.4 A 20nm 6gb function-in-memory dram, based on HBM2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'21)*, pages 350–352. 2021

[40] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*, pages 740–753. 2019

[41] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS'99)*, pages 134–143. 1999

[42] Donghun Lee, Jinin So, Minseon Ahn, Jong-Geon Lee, Jungmin Kim, Jeonghyeon Cho, Oliver Rebholz, Vishnu Charan Thummala, Ravi Shankar JV, Sachin Suresh Upadhya, Mohammed Ibrahim Khan, and Jin Hyun Kim. Improving in-memory database operations with acceleration DIMM (axdimm). In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD'22)*, pages 2:1–2:9. ACM, 2022. https://doi.org/10.1145/3533737.3535093

[43] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. MEMTIS: efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, pages 17–34. 2023

[44] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich.

Pytorch-biggraph: A large scale graph embedding system. In *Proceedings of Machine Learning and Systems (MLSys'19)*. 2019

[45] Cong Li, Zhe Zhou, Yang Wang, Fan Yang, Ting Cao, Mao Yang, Yun Liang, and Guangyu Sun. PIM-DL: expanding the applicability of commodity dram-pims for deep learning via algorithm-system co-optimization. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, pages 879–896. 2024

[46] Cong Li, Zhe Zhou, Size Zheng, Jiaxi Zhang, Yun Liang, and Guangyu Sun. Specpim: Accelerating speculative inference on pim-enabled system via architecture-dataflow co-exploration. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, pages 950–965. 2024

[47] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce L. Jacob. Dramsim3: A cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters*, 19(2):110–113, 2020

[48] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*, pages 1754–1763. 2018

[49] Haifeng Liu, Long Zheng, Yu Huang, Jingyi Zhou, Chaoqiang Liu, Runze Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. Enabling efficient large recommendation model training with near CXL memory processing. In *Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'24)*, pages 382–395. 2024

[50] André Lopes, Daniel Castro, and Paolo Romano. PIM-STM: software transactional memory for processing-in-memory systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, pages 897–911. 2024

[51] Haocong Luo, Yahya Can Tugrul, F. Nisa Bostanci, Ataberk Olgun, Abdullah Giray Yaglikçi, and Onur Mutlu. Ramulator 2.0: A modern, modular, and extensible DRAM simulator. *IEEE Computer Architecture Letters*, 23(1):112–116, 2024

[52] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R. Lyu, and Yangfan Zhou. CHIME: A cache-efficient and high-performance hybrid index on disaggregated memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP'24)*, pages 110–126. 2024

[53] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. SMART: A high-performance adaptive radix tree for disaggregated memory. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 553–571. 2023

[54] Microsoft. Best Practices on Recommendation Systems. https://github.com/recommenders-team/recommenders, 2025.

[55] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, pages 533–549. 2021

[56] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, K. R. Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA'22)*, pages 993–1011. 2022

[57] Antoine Murat, Clément Burgelin, Athanasios Xygkis, Igor Zablotchi, Marcos Kawazoe Aguilera, and Rachid Guerraoui. SWARM: replicating shared disaggregated-memory data in no time. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP'24)*, pages 24–45. 2024

[58] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman

Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019. https://arxiv.org/abs/1906.00091

[59] NVIDIA. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2025.

[60] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)*. 2023

[61] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. Product-based neural networks for user response prediction. In *Proceedings of the 16th IEEE International Conference on Data Mining (ICDM'16)*, pages 1149–1154. 2016

[62] Mohammad R. Rezaei. Amazon product recommender system. *CoRR*, abs/2102.04238, 2021. https://arxiv.org/abs/2102.04238

[63] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 69–87. 2018

[64] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. Ditto: An elastic and adaptive memory-disaggregated caching system. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, pages 675–691. 2023

[65] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A large-scale deep learning recommender system with low-latency model update. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, pages 821–839. 2022

[66] Harold S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, 19(1):73–78, 1970. https://doi.org/10.1109/TC.1970.5008902

[67] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, pages 105–121. 2023

[68] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. Towards high-throughput and low-latency billion-scale vector search via CPU/GPU collaborative filtering and re-ranking. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST'25)*, pages 171–185. 2025

[69] UPMEM. UPMEM DPU SDK Documentation. https://sdk.upmem.com/2025.1.0/, 2025.

[70] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*, pages 839–848. 2018

[71] Yitu Wang, Shiyu Li, Qilin Zheng, Linghao Song, Zongwang Li, Andrew Chang, Hai Li, and Yiran Chen. NDSEARCH: accelerating graph-traversal-based approximate nearest neighbor search through near data processing. In *Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'24)*, pages 368–381. 2024

[72] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-exclusive memory tiering via transactional page migration. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 19–35. 2024

[73] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: an efficient GPU embedding cache for personalized recommendations. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys'22)*, pages 402–416. 2022

[74] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. Gl-cache: Group-level learning for efficient and high-performance caching. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 115–134. 2023

[75] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex M. Bronstein, Ronald G. Dreslinski, Trevor N. Mudge, and Nishil Talati. GRACE: A scalable graph-based approach to accelerating recommendation model inference. In *Proceedings of the 28th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), pages 282–301. 2023

[76] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. FAERY: an fpga-accelerated embedding-based retrieval system. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, pages 841–856. 2022

[77] Junjie Zhang, Ruobing Xie, Yupeng Hou, Wayne Xin Zhao, Leyu Lin, and Ji-Rong Wen. Recommendation as instruction following: A large language model empowered recommendation approach. *CoRR*, abs/2305.07001, 2023

[78] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling multi-versioning for distributed transactions on disaggregated memory. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 801–819. 2024

[79] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: fast one-sided rdma-based distributed transactions for disaggregated persistent memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, pages 51–68. 2022

[80] Xiangyu Zhao, Haochen Liu, Hui Liu, Jiliang Tang, Weiwei Guo, Jun Shi, Sida Wang, Huiji Gao, and Bo Long. Autodim: Field-aware embedding dimension searchin recommender systems. In *Proceedings of the Web Conference 2021 (WWW'21)*, pages 3015–3022. 2021

[81] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 193–210. 2024

[82] Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*, pages 1059–1068. 2018

[83] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. Neomem: Hardware/software co-design for cxl-native memory tiering. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO'24)*, pages 1518–1531. 2024

A Artifact Appendix

Abstract

The CloverRec artifact contains the source code, build helpers, workload launchers, and result parsers for CloverRec and the three baselines evaluated in this paper. It supports role-by-role functional checks as well as automated end-to-end experiments on a user-managed GPU and UPMEM cluster. The artifact received the Artifacts Available, Artifacts Functional, and Results Reproduced badges.

Scope

The artifact enables validation of the paper's principal end-to-end claim: offloading embedding operations to a disaggregated PIM pool, together with CloverRec's dimension-parallel execution, adaptive hot-embedding replication, and rank-level transfer, improves recommendation-inference throughput and latency relative to local embedding, remote embedding, and a naive PIM baseline. The supplied launchers exercise the four systems using the synthetic RM1–RM4 configurations and the optional Criteo-Kaggle workload. Functional tests can run on smaller compatible configurations, whereas reproducing the reported absolute performance requires hardware comparable to the platform described in §5.1.

Contents

The repository contains four implementation directories: `cloverrec/`, `naive_pim_emb/`, `remote_emb/`, and `local_emb/`. The `scripts/` directory provides environment checks, native builds, role-specific smoke runs, automated experiment matrices, and result parsing. The Conda and pip dependency specifications are provided separately for the GPU/coordinator hosts and the PIM host. Synthetic RM1–RM4 workloads require no external dataset. The optional Criteo-Kaggle data is not redistributed and must be obtained and prepared separately.

Hosting

The artifact is publicly available at https://github.com/LighT-chenml/CloverRec. The frozen FAST '27 artifact is commit `4825b010d06c8347c713255c3d48c92473406bb0`, available directly at https://github.com/LighT-chenml/CloverRec/tree/4825b010d06c8347c713255c3d48c92473406bb0. The convenience branch `artifact/fast27` points to this commit. The commit records the public, user-managed-cluster instructions; the implementation and evaluation scripts are unchanged from the version evaluated by the Artifact Evaluation Committee.

Requirements

CloverRec uses three logical roles: a CUDA-capable model server, a coordinator, and an RDMA-connected embedding pool. The model and coordinator may share one host. Reproducing the paper's results requires a 100 Gbps RDMA network, an NVIDIA V100-class GPU, and a UPMEM server with eight PIM DIMMs (1020 functional DPUs in our platform). The evaluated software environment uses Ubuntu 22.04, Python 3.10, a compatible NVIDIA/CUDA stack, Mellanox `ibverbs`/`pyverbs`, and the UPMEM SDK. Passwordless SSH from the coordinator to each remote role is required by the automated launcher.

Getting Started

Clone the frozen commit into the same path on all participating hosts, create the environments from `environment.yml` and `environment-pim.yml`, and run `scripts/check_env.sh` for the relevant role on each host. Configure the model and embedding-pool RDMA addresses, remote SSH targets, repository paths, and Python interpreters as documented in `README.md`. Then invoke `scripts/run_e2e.py` with the `smoke` batch profile for RM1. This builds the required components, starts all roles, executes one representative point for each selected system, terminates remote processes, and writes logs and `summary.csv` under `results/e2e/`. After the environments are installed, this check normally completes within a few minutes on the evaluated cluster.

Evaluation Workflow

For the complete synthetic evaluation, run the end-to-end launcher for `cloverrec`, `remote_emb`, `naive_pim_emb`, and `local_emb` over RM1–RM4 with the `knee` batch profile. The launcher records throughput, average latency, embedding lookup time, and available breakdown fields for every configuration. The resulting CSV enables direct comparison of the four systems and validates the principal performance trends reported in the paper. On our configured cluster, the RM1–RM4 matrix completes in approximately one hour after the first build. The Criteo-Kaggle path is optional and requires the separately prepared dataset; because each coordinator run reloads the processed data, it can substantially increase total runtime. Detailed commands, parameter overrides, expected outputs, and failure diagnostics are maintained in the frozen `README.md`.