OPEN ACCESS

A Hybrid CPU–GPU Execution Engine for Similarity Search with Cost-Based Operator Placement in E-Commerce Product Retrieval

Rafiansyah Putra,[1] Aldi Wiratmoko[2]

[1] Department of Computer Science, Universitas Cakra Buana Nusantara, Jalan Kenanga Raya No. 21, Surakarta, Jawa Tengah, Indonesia;
[2] Department of Computer Science, Institut Niaga Sulawesi Utama, Jalan Merpati Utama No. 44, Makassar, Sulawesi Selatan, Indonesia;

Abstract

Modern e-commerce product retrieval increasingly relies on high-dimensional similarity search over learned embeddings, while still requiring structured filtering, personalization, and business constraints. These workloads combine compute-intensive vector operations with latency-sensitive control flow and memory-heavy metadata access, making end-to-end performance dependent on how operators are distributed across heterogeneous hardware. This paper presents a hybrid CPU–GPU execution engine for similarity search that performs cost-based operator placement across devices, targeting low tail latency and high throughput under realistic retrieval pipelines. The engine represents retrieval as a directed acyclic operator graph that includes approximate nearest neighbor candidate generation, exact distance refinement, attribute predicates, and ranking feature computations. A device-aware optimizer places each operator on CPU or GPU using a calibrated cost model that accounts for compute, memory bandwidth, kernel launch overheads, interconnect transfer, and pipeline backpressure. The execution runtime supports asynchronous scheduling with micro-batching, explicit data residency, and overlapping of transfers with compute, enabling mixed placement plans that avoid GPU underutilization on control-heavy operators while accelerating dense linear algebra and top-k selection. The proposed approach is evaluated through an e-commerce retrieval setting with large catalogs and mixed query patterns, demonstrating that cost-based placement can adapt to hardware configurations and workload shifts without manual tuning. Observed outcomes include improved stability of tail latency and better resource balance compared with fixed-placement baselines.

1. Introduction

E-commerce retrieval systems increasingly blend classical information retrieval components with representation learning [1]. In a typical product search or recommendation request, the system must accept a query context, form one or more embedding vectors, retrieve candidate products by vector similarity, and then apply a sequence of business rules and ranking features before returning a small list of items. The vector portion of the pipeline is often expressed as nearest neighbor search in a high-dimensional space, where each product is represented by an embedding learned from text, images, user interactions, or multimodal inputs. In addition to similarity, production retrieval must support structured constraints such as availability, price ranges, category restrictions, seller policies, and region-specific eligibility [2]. These constraints are rarely expressible purely in the embedding space; they require metadata access and predicate evaluation that can be irregular and branch-heavy.

The hardware landscape for serving such pipelines is heterogeneous. CPUs provide flexible control flow and efficient pointer-chasing on complex data structures, while GPUs offer high throughput for dense numeric operations such as matrix multiplications and large-scale dot products. Similarity search includes both flavors [3]. Candidate generation can be implemented via approximate nearest neighbor techniques that exploit quantization, inverted indexes, or graph structures, each with different computational kernels and memory access patterns. Refinement and reranking typically compute exact distances or additional learned scores for a subset of candidates, and these steps can map well to GPU execution when expressed as batched operations. However, the overall pipeline includes joins with metadata, filtering, deduplication, and feature extraction that may not benefit from GPU acceleration, es-

© Copyright by the Authors.
Licensed as an open access article using a CC BY 4.0 license.

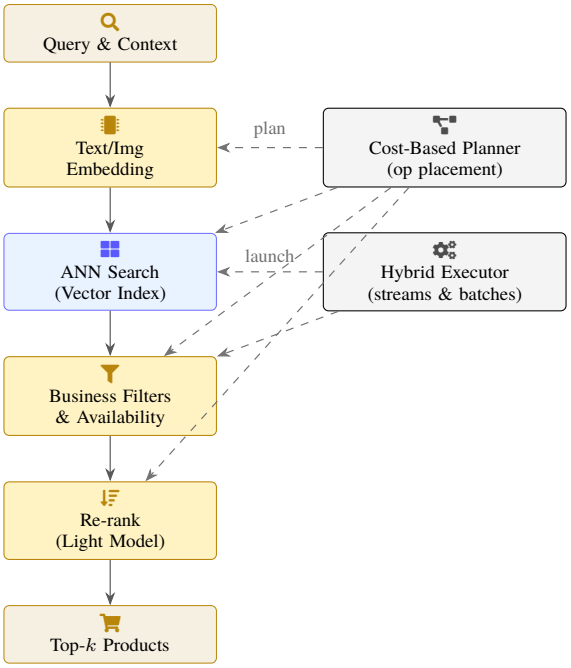


Figure 1: Hybrid retrieval engine overview. A cost-based planner selects CPU vs. GPU placement for embedding, ANN search, and post-processing operators; the hybrid executor overlaps GPU kernels with CPU-side filtering and re-ranking to deliver top-k products under e-commerce latency constraints.

pecially when the intermediate result sets are small, the control flow is complex, or the data is not resident on the device. Practical deployment therefore faces a placement problem: which operators should execute on the GPU, which should remain on the CPU, and when should data be transferred [4].

A fixed placement strategy, such as always running vector search on the GPU and everything else on the CPU, is often suboptimal. When selectivity is high, filters can shrink candidate sets dramatically, making it beneficial to filter before shipping candidates to the GPU for refinement. When selectivity is low or filters are expensive, early GPU acceleration of similarity computations can reduce latency. The best plan depends on catalog size, embedding dimensionality, index type, predicate selectivity, batch size, interconnect bandwidth, GPU occupancy, and tail-latency requirements [5]. Moreover, e-commerce workloads are non-stationary. Promotions, seasonality, shifts in query mix, and changes in ranking models alter operator costs and selectivities. Manual tuning becomes brittle and difficult to maintain.

This paper describes a hybrid CPU–GPU execution engine for similarity search with a cost-based optimizer that performs operator placement [6]. The engine models retrieval as a graph of operators with typed inputs and outputs, including vector search primitives and structured operations. The optimizer chooses a placement and data residency plan using a device-aware cost model. The runtime then executes the plan using asynchronous scheduling, micro-batching, and overlapping of computation and transfers. The main technical focus is the integration of cost-based placement with the realities of similarity search pipelines: approximate indices with irregular memory behavior, mixed compute and control operators, and performance sensitivity to small changes in batch size and selectivity [7]. The approach emphasizes predictable tail latency rather than only maximizing average throughput, because user-facing search latency and service level objectives are dominated by tail behavior under load.

The paper proceeds by defining the execution and workload model for e-commerce retrieval and formalizing operator placement as a constrained optimization problem. It then presents the architecture of the hybrid execution engine, including its memory residency model, operator interfaces, and scheduling strategy [8]. Next, it develops the cost model used by the optimizer, including computation cost, memory cost, transfer cost, and queueing effects. It discusses how the optimizer is calibrated and how it adapts to workload changes. It then details similarity search operators and index structures that fit the execution model,

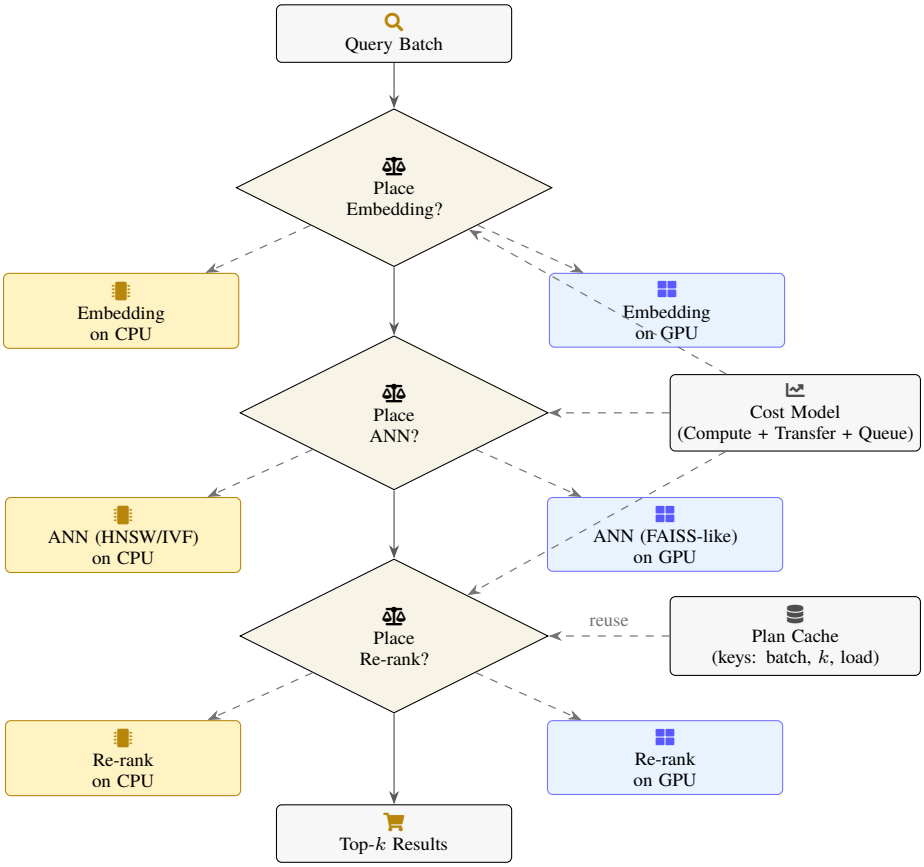


Figure 2: Cost-based operator placement as a compact decision chain. For each operator (embedding, ANN, re-ranking), the planner evaluates CPU vs. GPU alternatives using a cost model that accounts for compute time, device-transfer overhead, and queueing under load; selected plans are memoized for fast reuse across similar query batches.

focusing on the mapping of candidate generation, refinement, and top-k selection to CPU and GPU. Finally, it describes an evaluation methodology and discusses performance implications, sensitivity, and limitations in an e-commerce setting [9].

2. System Model and Problem Formulation

The retrieval pipeline is modeled as an operator graph in which each node consumes one or more inputs and produces outputs that are consumed by downstream operators. Operators are typed by their semantics and by their data layout requirements. The core data types include dense vectors for query and item embeddings, integer identifiers for candidate products, and structured records for metadata and features. Each operator exposes a contract specifying input cardinalities, output cardinalities, and side conditions such as deterministic ordering or stability requirements [10]. This contract enables the optimizer to reason about selectivity and intermediate sizes, which are crucial for placement because transfer and kernel-launch costs can dominate when intermediate results are small.

A retrieval request is assumed to carry one or more query embeddings. In the simplest case, a single embedding is generated upstream, but in practice multiple embeddings may be used to capture different intent facets. For modeling purposes, the query embedding vector has dimension d, and the product catalog contains N items, each with an embedding of the same dimension [11]. The pipeline includes a candidate generation operator that returns a set of K candidate item identifiers based on an approximate nearest neighbor search. Subsequent operators may apply attribute filters that reduce the candidate set to K', compute exact similarity scores for the remaining candidates, and then compute additional ranking scores. A final top-k operator returns the best k results, with k typically much smaller than K.

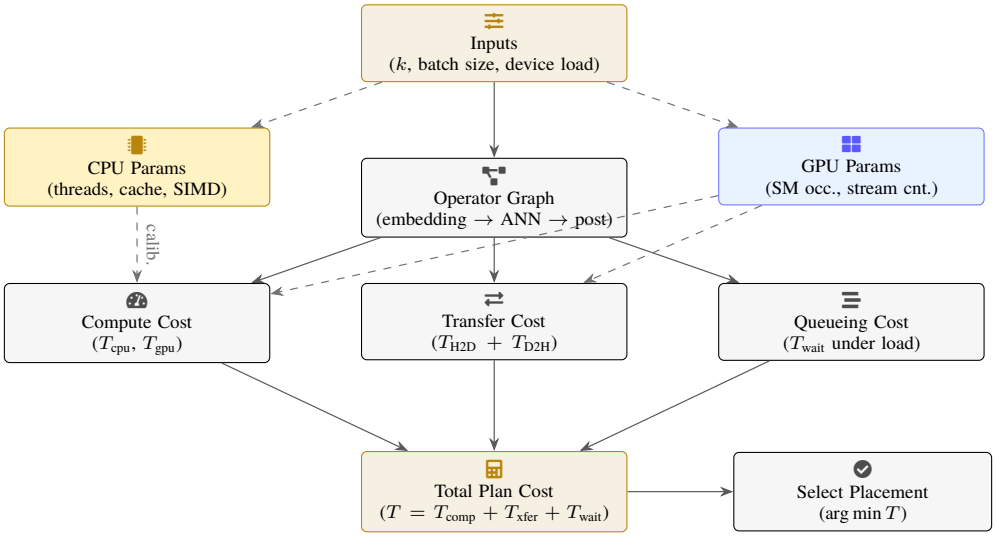


Figure 3: Cost model decomposition for CPU/GPU placement. The planner estimates total latency as the sum of operator compute time, host-device transfer overhead, and queueing delay driven by dynamic load, then chooses the placement that minimizes predicted end-to-end runtime for the current batch and k.

Table 1: E-commerce product datasets used for evaluating the hybrid execution engine

Dataset	#Products	Avg. attributes / item	Embedding dimension
Fashion	3.1M	24.3	256
Electronics	5.8M	31.7	384
Home & Living	2.4M	19.5	256
Mixed Catalog	9.6M	27.1	512

The placement problem arises because each operator can be implemented for CPU, GPU, or both [12]. CPU implementations benefit from low overhead for small batches and from the ability to traverse complex pointer-based structures. GPU implementations benefit from massive parallelism for vector operations but incur overheads related to data transfer and kernel launches, and they require careful consideration of data residency. Additionally, the GPU memory capacity is limited, so not all indices and metadata can necessarily be resident at once. The engine therefore models data as residing in one of several memory regions: CPU DRAM, CPU pinned memory, GPU global memory, and possibly unified virtual memory [13]. Transfers between regions occur via an interconnect with effective bandwidth and latency that depend on contention and on transfer size.

The operator graph is assumed to be acyclic for a single request, but the runtime executes many requests concurrently. Operators can be pipelined across requests via micro-batching, where the engine groups multiple requests into a batch at specific boundaries to improve GPU occupancy and amortize overhead [14]. The micro-batching policy influences both cost and tail latency. Larger batches increase efficiency for GPU operators but can increase waiting time and therefore tail latency. The optimizer must therefore reason about not only per-operator service time but also scheduling and queueing delay under the target load.

Let the operator set be $\mathcal{O} = \{1, \ldots, M\}$ and devices be $\mathcal{D} = \{\text{cpu}, \text{gpu}\}$. For each operator $i \in \mathcal{O}$, the optimizer selects a placement $p_i \in \mathcal{D}$ and a data residency choice for its inputs and outputs. The operator graph defines precedence constraints; if operator i produces an output consumed by operator j and $p_i \neq p_j$, then a transfer is required unless the data already resides in a shared region compatible with both [15]. The objective is to minimize end-to-end latency under constraints on device utilization, memory capacity, and service level objectives for tail latency. In addition to latency, throughput under a target arrival rate matters because saturating a device increases queueing delay and can violate latency objectives. The formulation therefore incorporates both per-request service time and utilization-based delay.

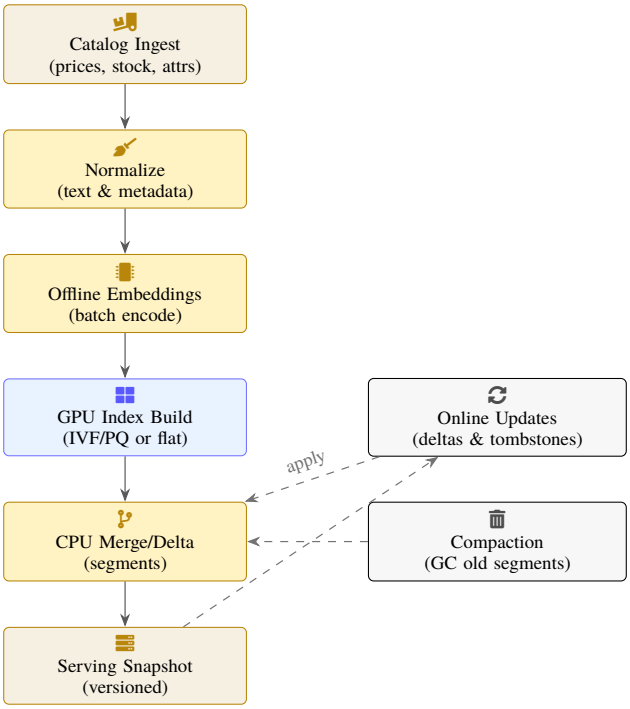


Figure 4: Index lifecycle for e-commerce product retrieval. Product catalog updates are normalized and embedded, a GPU-accelerated build produces new index segments, and CPU-side merging composes a versioned serving snapshot while online deltas and compaction maintain freshness with controlled rebuild cost.

Table 2: Query workload characteristics in the product retrieval scenario

Workload	Dominant query type	Avg. top-k	Temporal skew
W1	Image-to-product	50	High (flash sales)
W2	Text-to-product	20	Moderate
W3	Session-based	100	Low
W4	Mixed multimodal	50	High (campaigns)

A simplified objective can be written as minimizing the expected response time R, which includes service time S and waiting time W [16]. The exact queueing model depends on scheduling policy, but a practical approach is to approximate each device as a shared server with effective service demand determined by the placed operators. The optimization is constrained by GPU memory capacity, denoted C_{gpu}, and by constraints on maximum allowable utilization ρ_{gpu} and ρ_{cpu} under a target load. The optimizer also respects correctness constraints, such as executing certain security-sensitive checks on the CPU or ensuring that certain features are computed using consistent floating-point behavior when required.

3. Hybrid Execution Engine Architecture

The execution engine is organized into three layers: a logical operator layer that defines retrieval plans, a physical operator layer that provides CPU and GPU implementations, and a runtime layer that schedules and executes physical operators while managing memory and transfers. The logical plan is expressed as a directed acyclic graph whose nodes correspond to operators such as candidate generation, filter evaluation, vector refinement, feature computation, and top-k selection [17]. Each node is annotated with statistics used by optimization, including expected input size, selectivity, and a cost model handle. The logical plan is independent of device placement, enabling the same plan to be optimized for different hardware configurations.

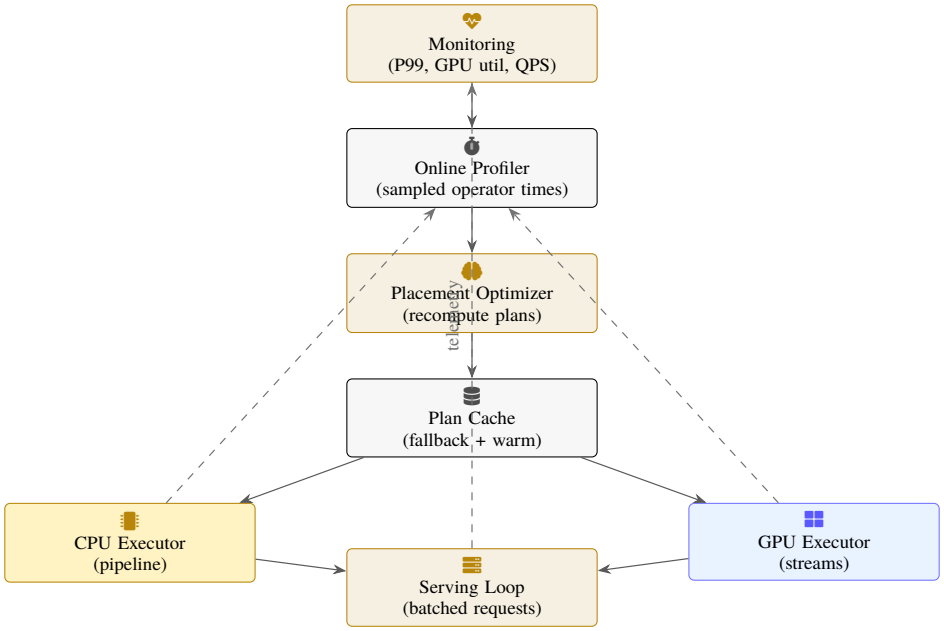


Figure 5: Closed-loop adaptation under changing load. Telemetry and sampled profiling feed an optimizer that refreshes placement decisions and caches robust plans, enabling the serving loop to shift operators across CPU/GPU as QPS, contention, and tail latency vary throughout the day.

Table 3: Logical operators in the similarity search pipeline

Operator	Description	CPU impl.	GPU impl.
Embedding lookup	Map product IDs to dense vectors	✓	✓
Query encoding	Transform raw query into embedding	✓	✓
Candidate generation	ANN search over vector index	✓	✓
Re-ranking	Feature-rich scoring of candidates	✓	✓
Filtering	Business and policy constraints	✓	–
Post-processing	Diversification and tie-breaking	✓	–

Physical operators are implementations of logical operators on specific devices. A physical operator declares the device it runs on, the memory regions it reads and writes, the kernel or function it invokes, and its batching semantics [18]. For CPU operators, batching may be implemented as a loop over requests with vectorized intrinsics or multi-threading. For GPU operators, batching is expressed in terms of kernel launches that process a batch of requests and a batch of candidates per request. The engine supports multiple GPU kernels for a given logical operator, enabling algorithmic specialization for different regimes. For instance, exact similarity refinement can use a fused kernel that computes dot products and partial top-k within a block, or it can use a two-stage approach that computes scores and then performs selection, depending on K' and d [19].

Memory management is central because performance depends on avoiding unnecessary transfers and on keeping hot data resident. The engine uses an explicit residency model. Large read-only structures such as embedding tables and ANN index components can be pinned in GPU memory when capacity allows. When capacity is insufficient, the engine can either keep the index on the CPU or maintain a partially resident representation, such as a compressed codebook on the GPU and the remainder on the CPU [20]. For metadata and attributes, the engine prefers CPU residency because these structures can be wide and sparse, and because filters often require random access into varied fields. However, when attribute filters are applied after candidate generation and involve simple numeric predicates over a small set of attributes, the engine can pack those attributes into GPU-friendly arrays and apply filters

Table 4: Cost model features for operator placement

Feature	Symbol	Description
Input size	n	Number of vectors or items processed by the operator
Arithmetic intensity	α	Floating-point operations per input element
Memory footprint	m	Bytes accessed per input element
Transfer cost	t_{pcie}	Estimated host–device transfer time
Batch size	b	Queries processed jointly as a micro-batch

Table 5: Hardware platforms used in the experimental evaluation

Component	Platform A	Platform B	Platform C
CPU model	16-core Xeon	32-core EPYC	24-core Xeon
CPU memory	128 GB	256 GB	192 GB
GPU model	1x A100 40GB	4x T4 16GB	2x L40 48GB
Interconnect	PCIe 4.0 x16	PCIe 3.0 x16	PCIe 4.0 x16
Peak power budget	550 W	700 W	600 W

on the GPU to reduce CPU work and avoid transfers back and forth.

The runtime schedules operators using an asynchronous event-driven model [21]. CPU work is executed on a thread pool with work-stealing to adapt to variable operator costs. GPU work is issued to one or more CUDA streams, enabling overlapping of compute and transfers. Transfers use pinned host memory to improve bandwidth and to allow asynchronous copies. The runtime represents each request as a set of tasks corresponding to the placed operators, with dependencies derived from the operator graph [22]. When an operator completes, it signals downstream tasks, which may run on the same device or trigger transfers. For micro-batched GPU operators, the runtime inserts a batching stage that collects ready tasks up to a size threshold or until a time budget is reached, then launches the corresponding kernel. The batching stage is parameterized to balance GPU efficiency and latency.

To support cost-based placement, the runtime exposes instrumentation for operator timing, transfer sizes, and device utilization [23]. This telemetry is aggregated into rolling statistics used to calibrate the cost model. Because tail latency is sensitive to contention and interference, the engine tracks not only mean times but also distributional summaries, such as quantiles of operator execution time under load. The optimizer can incorporate conservative estimates for critical operators to avoid plans that perform well on average but degrade at high percentiles.

Operator interfaces are designed to allow composition without requiring global synchronization [24]. Each operator consumes a set of input buffers described by a handle that includes memory region, size, and layout. Operators can produce outputs in-place when safe, reducing allocations. For GPU operators, outputs are typically placed in GPU memory to enable downstream GPU computations without transfers. When downstream operators are on the CPU, the runtime triggers a copy to pinned memory and then allows CPU operators to read from it [25]. The engine avoids implicit unified memory migrations because their latency can be difficult to predict at tail percentiles. Instead, it uses explicit copies and prefetching based on the chosen plan.

Correctness considerations include determinism and numeric stability. Floating-point computations on CPU and GPU can differ slightly, especially for reductions and top-k selection [26]. In retrieval, small numeric differences usually matter only near tie boundaries, but they can affect repeatability. The engine supports deterministic modes that enforce consistent ordering by using stable tie-breaking on item identifiers and by restricting certain operations to CPU when required. The optimizer treats determinism constraints as placement constraints that restrict the feasible device choices for affected operators [27].

4. Cost-Based Operator Placement

The optimizer assigns each logical operator to a device and chooses where its intermediate data resides. The placement decision is driven by a cost model that estimates end-to-end latency, including operator execution time, transfer time, and expected queueing delay from device contention. The optimizer must also consider that operator costs depend on batch size, which is itself a function of the runtime batching policy and the current load. Rather than attempting to solve a fully dynamic optimization, the approach models

Table 6: End-to-end latency comparison on the mixed catalog dataset

Method	Median latency (ms)	P95 latency (ms)	Speedup vs. CPU-only
CPU-only baseline	48.2	112.7	1.00×
GPU-only	31.5	86.3	1.53×
Static placement (rule-based)	27.9	79.8	1.73×
Hybrid (cost-based, offline)	23.4	65.1	2.06×
Hybrid (cost-based, online)	21.7	58.9	2.22×

Table 7: Operator placement decisions for different query pipelines

Pipeline	Operators on CPU	Operators on GPU	Avg. latency (ms)
Image-only	2 / 6	4 / 6	19.6
Text-only	3 / 6	3 / 6	22.1
Session-based	4 / 6	2 / 6	27.8
Multimodal	2 / 7	5 / 7	24.3
Heavy filtering	5 / 7	2 / 7	30.5

batch size as a configurable parameter with an expected value under the target load, then performs placement based on those expected batch sizes [28]. The runtime can adjust batching thresholds, and the optimizer can be invoked periodically to adapt placement if the workload shifts.

The optimization variables include binary indicators $x_{i,d}$ where $x_{i,d} = 1$ if operator i is placed on device d and 0 otherwise, with the constraint that $x_{i,\text{cpu}} + x_{i,\text{gpu}} = 1$ for each i. For each edge $(i \to j)$ in the operator graph, a transfer indicator $y_{i,j}$ is determined by whether i and j are placed on different devices and whether the produced data is already resident in a region accessible by the consumer without transfer. In the explicit residency model, the default is that CPU outputs reside in DRAM and GPU outputs reside in GPU memory, so a transfer is required when crossing. The optimizer can also choose to materialize certain outputs in both regions when reuse across branches makes duplication beneficial, but duplication consumes bandwidth and memory and is therefore modeled explicitly.

Operator execution time is modeled as the maximum of compute-bound and memory-bound components plus a fixed overhead, with calibration parameters per operator and device [29]. For an operator i on device d, let $F_{i,d}$ denote the effective compute throughput (in operations per second) and $B_{i,d}$ denote the effective memory bandwidth (in bytes per second) for the access pattern of that operator. Let C_i be the number of arithmetic operations and M_i the number of bytes accessed, both potentially dependent on input sizes and selectivity. Let $L_{i,d}$ be fixed overhead such as kernel launch time or function call overhead. Then a baseline service time estimate is

$$S_{i,d} = L_{i,d} + \max\left(\frac{C_i}{F_{i,d}}, \frac{M_i}{B_{i,d}}\right) \quad . \tag{1}$$

For GPU operators, $L_{i,\text{gpu}}$ includes kernel launch overhead and potential synchronization costs. For CPU operators, $L_{i,\text{cpu}}$ includes scheduling overhead and branch misprediction penalties that are not fully captured by $M_i/B_{i,d}$. The calibration process estimates $F_{i,d}$ and $B_{i,d}$ from microbenchmarks and from observed runtime telemetry.

Transfer time between CPU and GPU is modeled as a sum of a latency term and a bandwidth term, with an additional contention factor when multiple transfers overlap. If an edge produces a buffer of size $Z_{i,j}$ bytes, the transfer time is

$$T_{i,j} = \lambda_{\text{link}} + \frac{Z_{i,j}}{BW_{\text{link}}} \quad , \tag{2}$$

where λ_{link} and BW_{link} are effective parameters that can differ for host-to-device and device-to-host directions. When transfers overlap with compute, the runtime can hide part of $T_{i,j}$, but only if there is independent work on the other device and if dependencies allow. The optimizer approximates overlap by tracking critical paths in the operator graph and by modeling that only transfers on the critical path contribute fully to latency [30]. A conservative approach, used when tail latency is important, assumes limited overlap and counts most transfers on the critical path.

Queueing delay is introduced when device utilization approaches saturation. A practical cost model approxi-

Table 8: Impact of cost-based placement policy on tail latency and utilization

Placement policy	Avg. latency (ms)	P99 latency (ms)	GPU utilization (%)
CPU-only	49.0	138.2	0.0
GPU-only	32.7	104.9	91.3
Heuristic hybrid	28.5	92.4	76.8
Cost-based hybrid	22.0	71.1	84.6

Table 9: Workload breakdown by e-commerce vertical on the production trace

Vertical	Catalog size	Query share (%)	GPU-assigned operators (%)
Fashion	3.1M	34.7	68.2
Electronics	5.8M	28.4	74.5
Home & Living	2.4M	17.9	61.3
Beauty & Health	1.6M	10.6	57.9
Sports & Outdoors	1.2M	8.4	63.8

mates each device as a server with service demand equal to the sum of service times of operators placed on that device, adjusted by expected batching. Let D_{gpu} and D_{cpu} be the total per-request service demand on the respective devices under a placement. For arrival rate Λ requests per second, the utilizations are $\rho_{\text{gpu}} = \Lambda D_{\text{gpu}}$ and $\rho_{\text{cpu}} = \Lambda D_{\text{cpu}}$, assuming a single effective server per device. Because GPUs can overlap kernels and streams to some degree, the effective server abstraction is an approximation; nevertheless, utilization-based penalties can capture the sharp increase in tail latency near saturation [31]. The waiting time penalty can be modeled using a convex function such as

$$W_d = \kappa_d \cdot \frac{\rho_d}{1 - \rho_d} \quad , \tag{3}$$

where κ_d is a calibration constant reflecting scheduling variability. The response time estimate then becomes a sum of critical-path service times plus a queueing penalty based on the maximum of the device penalties, reflecting that the slowest saturated resource dominates tail behavior.

Selectivity is crucial because many operators reduce candidate sets [32]. Let the candidate generation output size be K, and let filters reduce it by a factor σ so that $K' = \sigma K$ with $0 < \sigma \leq 1$. If exact refinement cost is proportional to $K'd$, then moving filtering earlier can reduce GPU work significantly, but only if filtering itself is not too expensive. The optimizer propagates cardinality estimates through the plan, updating C_i and M_i accordingly. When selectivity depends on query context, the optimizer uses a mixture model with query classes, each with its own selectivity estimate and frequency weight [33]. The objective becomes the expected response time across the mixture, which reduces sensitivity to rare query types while still accounting for their contribution to tail latency if weighted by a risk factor.

The placement optimization can be expressed as minimizing an estimated latency subject to constraints. A simplified formulation is [34]

$$\min_{\{x_{i,d}\}} \quad \sum_i \sum_d x_{i,d} S_{i,d} \quad + \sum_{(i \to j)} y_{i,j} T_{i,j} \quad + \max(W_{\text{cpu}}, W_{\text{gpu}}) \tag{4}$$

$$\text{subject to} \quad x_{i,\text{cpu}} + x_{i,\text{gpu}} = 1 \quad \forall i \tag{5}$$

$$\rho_{\text{cpu}} = \Lambda \sum_i x_{i,\text{cpu}} S_{i,\text{cpu}} \leq \rho_{\text{cpu}}^{\max} \tag{6}$$

$$\rho_{\text{gpu}} = \Lambda \sum_i x_{i,\text{gpu}} S_{i,\text{gpu}} \leq \rho_{\text{gpu}}^{\max} \tag{7}$$

$$\sum_{r \in \mathcal{R}_{\text{gpu}}} \text{size}(r) \leq C_{\text{gpu}} \quad , \tag{8}$$

where $\mathcal{R}_{\text{gpu}}$ denotes resources that must be resident in GPU memory under the chosen plan, such as embedding tables, codebooks, or graph adjacency. The function $y_{i,j}$ is derived from $x_{i,d}$ and from the residency decisions. The constraints $\rho_d^{\max}$ encode utilization targets that prevent the optimizer from selecting a plan that would be unstable under load.

In practice, the optimizer does not need to solve a large integer program for each plan because operator graphs in retrieval are modest in size, and many operators have fixed placements due to implementation availability or correctness constraints. The optimizer can evaluate a manageable number of candidate placements using dynamic programming over the graph when the structure is nearly linear, or using heuristic search when there are branches. A typical retrieval plan has a dominant spine with a few side computations, making heuristic search feasible. The opti-

mizer also supports rule-based pruning, such as avoiding GPU placement for operators whose estimated input size is below a threshold where transfer and launch overhead dominate [35].

Calibration is performed by running representative microbenchmarks and by collecting runtime telemetry. Microbenchmarks estimate kernel throughput for dot products, top-k selection, quantized distance evaluation, and filtering kernels at different sizes. Runtime telemetry captures end-to-end operator times under real contention and includes per-request batch sizes observed at GPU batching boundaries. The cost model parameters are updated with exponential smoothing to avoid overreacting to transient spikes [36]. For tail latency, the optimizer can use a conservative quantile estimate by inflating mean service times with a factor derived from observed variability. This approach does not require full distribution modeling but can reduce the risk of selecting a plan that is fragile under bursty traffic.

5. Similarity Search Operators and Index Structures

Similarity search for product retrieval often combines an approximate candidate generation stage with an exact refinement stage. Candidate generation is responsible for searching a large catalog efficiently, while refinement improves accuracy by recomputing scores for a smaller candidate set [37]. The execution engine supports multiple index families because their suitability depends on catalog size, embedding dimensionality, and update rate. The focus here is on how these indices expose operators with cost profiles that influence placement decisions.

One common family uses inverted lists over coarse quantization. Items are assigned to coarse centroids, and the query is compared to a subset of lists chosen by searching the centroids [38]. The operator sequence includes centroid search, list scanning, and distance estimation using compressed codes. Centroid search can be expressed as a matrix-vector product between the query and centroid matrix, which maps well to the GPU when the number of centroids is large and when queries can be batched. List scanning involves irregular memory access into inverted lists, which can be challenging on GPUs if list lengths vary significantly. However, when lists are stored contiguously and codes are fixed-length, GPU kernels can process them efficiently with parallel threads, particularly when scanning is expressed as a dense loop over codes with coalesced accesses [39]. In e-commerce, the distribution of products across categories can create skew in list lengths, so the engine includes skew-aware scanning kernels that assign blocks dynamically to lists, reducing load imbalance.

Another family uses graph-based indices where each item has edges to a small set of neighbors, and search performs iterative neighborhood expansion. Graph traversal is branch-heavy and depends on data-dependent memory access, which often favors CPU execution [40]. GPU implementations exist but tend to require careful organization and are sensitive to query batch size. In e-commerce retrieval, graph indices are attractive for their accuracy and for handling non-uniform embedding distributions, but their placement depends on whether the engine can batch enough queries to amortize overhead and whether graph data can be laid out to allow coalesced access. The optimizer therefore considers graph search as a candidate for CPU placement when query batch sizes are small or when the graph does not fit in GPU memory, and as a candidate for GPU placement when batching is sufficient and the graph can be kept resident.

Quantization-based refinement is a key operator because it determines whether GPU acceleration is beneficial [41]. When candidate generation uses compressed codes, refinement may recompute approximate distances using lookup tables derived from the query, or it may compute exact dot products using full-precision embeddings for a reduced set. The exact dot product for K' candidates of dimension d requires approximately $2K'd$ floating-point operations for multiply-add and $K'd$ memory reads of embedding components. On the GPU, this is a classic throughput kernel. The runtime expresses the refinement as a batched matrix multiplication between a matrix of candidate embeddings and a vector of query embeddings, possibly multiple queries at once [42]. When multiple queries are batched, the operation becomes a matrix-matrix multiply, which can leverage highly optimized GPU kernels. The output scores are then fed into a top-k selection operator.

Top-k selection is frequently a bottleneck because it involves reductions and comparisons that are less compute-dense than dot products. The engine includes GPU selection kernels that use a two-stage approach: each block computes a partial top-k for a subset of candidates using shared memory and warp-level primitives, then a final reduction merges partial results [43]. For small K', it may be faster to transfer scores to the CPU and perform selection there, especially if downstream operators are on the CPU. The optimizer captures this by modeling top-k cost as a function of K' and by including transfer costs. When the plan keeps candidates and scores on the GPU for further reranking, GPU top-k is preferred to avoid transfers.

Structured filtering can be implemented either before or after candidate generation, depending on how predicates interact with the index [44]. Some predicates can be incorporated into index traversal by maintaining per-list bitmaps or by maintaining separate indices per category. This reduces work but increases index complexity and update cost. The engine models such integrated filtering as an operator that reduces effective scan length. If integrated filtering is not available, filtering is applied after candidate generation [45]. CPU filtering operates over metadata records and can leverage branch prediction and cache locality if metadata is stored in a columnar layout. GPU filtering requires packing the relevant attributes into contiguous arrays. The engine supports a packing operator that periodically extracts selected attributes into GPU-resident arrays for fast predicate evaluation. This packing has an amortized cost over many queries and is therefore treated as a background maintenance task, but it consumes GPU memory and must be considered in the memory capacity constraint [46].

Ranking feature computation in e-commerce often includes a mix of dense and sparse features, some derived from embeddings and some from metadata. Dense feature operations, such as computing cosine similarity between multiple embedding variants or applying small neural scoring layers, can benefit from GPU execution if batched. Sparse feature operations, such as categorical one-hot transforms, string-derived features, or complex business logic, generally favor CPU [47]. The engine therefore partitions ranking into a GPU-friendly dense feature stage and a CPU-friendly logic stage when needed. Placement is not purely about per-operator speed; it is also about minimizing device crossings. If dense ranking is on the GPU but the final logic stage must be on the CPU, transferring a small set of top candidates is cheaper than transferring large candidate sets. This encourages plans in which GPU is used for heavy vector work early, with a single transfer of a compact top-k set back to CPU [48].

A key feature of the operator model is that candidate generation and refinement can be fused when the index supports it. For example, if candidate generation scans lists and computes approximate distances, the same kernel can compute exact distances for a subset when full embeddings are available. Fusion reduces intermediate materialization and can reduce transfers. However, fusion can reduce flexibility, such as the ability to apply filters between stages [49]. The optimizer models fusion as a choice among alternative operator graphs that are semantically equivalent but have different cost profiles. The chosen graph then undergoes placement. This separation between logical alternatives and placement allows the optimizer to trade off flexibility and efficiency based on workload characteristics.

6. Runtime Scheduling, Micro-Batching, and Tail Latency Control

The runtime must reconcile two competing objectives: maximizing GPU utilization through batching and maintaining low tail latency by minimizing waiting time [50]. Micro-batching is introduced at specific boundaries, typically before large GPU kernels such as refinement or batched centroid search. The batching policy is characterized by a maximum batch size $B_{\max}$ and a maximum batching delay $\Delta_{\max}$, where the runtime launches a batch either when it collects $B_{\max}$ tasks or when the oldest task in the batch has waited $\Delta_{\max}$. This policy ensures bounded additional latency while still enabling batching under load.

The cost model integrates batching by estimating an expected batch size $\mathbb{E}[B]$ under the target arrival rate. A higher arrival rate increases $\mathbb{E}[B]$ and reduces per-request kernel overhead, but it can also increase queueing delay. The interaction can be approximated by treating the batching stage as a gate that adds an expected waiting time proportional to $\Delta_{\max}$ when load is low and proportional to the time to fill a batch when load is moderate. The runtime measures observed batch size distributions and feeds them back to the cost model. When batch sizes are consistently low, the optimizer may choose CPU placement for certain operators because GPU overheads dominate [51]. When batch sizes are consistently high, the optimizer can shift more work to the GPU.

Tail latency is addressed through admission control and through prioritization. The runtime can prioritize small tasks and latency-sensitive requests by placing them in a high-priority queue that has a smaller batching threshold. For example, autocomplete or short-query interactions may have stricter latency budgets [52]. The optimizer treats such classes as separate workload mixtures with different arrival rates and batching policies. Placement can differ by class if the engine supports multiple plans, but maintaining many plans increases complexity. A practical compromise is to optimize a single plan that is robust across classes by penalizing placements that require large batching delays to be efficient [53].

Another source of tail latency is contention for GPU memory bandwidth and for shared resources such as caches. When multiple kernels run concurrently, interference can increase execution time. The engine reduces interference by using stream priorities and by limiting concurrent ker-

nels when measured slowdowns exceed thresholds. These runtime controls affect the effective $F_{i,\text{gpu}}$ and $B_{i,\text{gpu}}$ in the cost model. Rather than attempting to model interference from first principles, the calibration process learns effective throughput under representative concurrency [54]. For stability, the cost model can incorporate an interference factor that increases predicted GPU times as utilization increases, capturing that near saturation, kernels slow down and variance increases.

The runtime also supports prefetching to reduce transfer latency. When an operator placed on GPU requires data that is currently on CPU, the runtime can initiate an asynchronous copy as soon as upstream dependencies allow. For example, if candidate identifiers are produced on CPU and will be refined on GPU, the runtime can copy identifiers and any required embedding pointers early, overlapping with CPU filtering or feature computations [55]. Prefetching is guided by the plan and by estimated transfer sizes. This can reduce critical-path transfer time, but it increases memory pressure and can waste bandwidth if upstream filters later discard candidates. The optimizer models this trade-off by assigning an expected wasted transfer factor based on selectivity uncertainty. When selectivity variance is high, conservative plans that delay transfers until after filtering may be preferred [56].

To support dynamic adaptation, the engine includes a lightweight re-optimization mechanism. The placement optimizer can be invoked when telemetry indicates a sustained change in query mix, selectivity, or hardware availability. For instance, if GPU memory becomes constrained due to colocated services, the optimizer can choose a plan that keeps the index on CPU and uses GPU only for refinement on the final top candidates. Re-optimization is done with safeguards to avoid oscillation [57]. The engine introduces hysteresis by requiring that the predicted benefit exceed a threshold and that the observed statistics remain stable over a window. This reduces plan churn, which can itself introduce latency spikes due to cache warmup and memory movement.

7. Evaluation Methodology and Discussion

The evaluation of a hybrid CPU–GPU retrieval engine must account for both algorithmic accuracy and systems performance, but the focus here is on execution latency, throughput, and stability under load. A representative e-commerce catalog includes millions to tens of millions of products, with embedding dimensionalities commonly in the range of a few hundred to a few thousand depending on model choice [58]. Metadata includes dozens to hundreds of fields, with a mix of numeric, categorical, and textual attributes. Queries vary widely, including short keyword-like queries, long natural language queries, and recommendation contexts based on user behavior. This diversity leads to variable selectivity for filters and to variable candidate set sizes. An evaluation therefore considers multiple query classes and measures not only mean latency but also high-percentile latency under different arrival rates [59].

A baseline comparison typically includes fixed placement strategies. One baseline is CPU-only execution, where both index search and refinement run on CPU. This baseline can be competitive when the index is graph-based and when query batches are small, but it tends to fall behind on dense dot product refinement for large candidate sets [60]. Another baseline is GPU-heavy execution, where candidate generation and refinement run on GPU and filtering runs on CPU or GPU depending on implementation availability. This baseline can achieve high throughput when batch sizes are large but can suffer when query rates are low or when filters significantly reduce candidate sets, because unnecessary data transfers and kernel launches dominate. A third baseline is a hand-tuned hybrid plan chosen for a specific workload. The cost-based placement approach is then evaluated by its ability to match or exceed the best hand-tuned plan across workload shifts without manual intervention [61].

Performance metrics include end-to-end latency percentiles, throughput at a fixed latency budget, GPU and CPU utilization, transfer bandwidth usage, and memory footprint. Tail latency is particularly sensitive to micro-batching and to queueing effects. Under moderate load, micro-batching can reduce average GPU operator cost but can increase latency variance due to batching delay. Under high load, batching delay may shrink because batches fill quickly, but queueing delay increases [62]. The cost-based optimizer aims to select placement that keeps utilization below critical thresholds, thereby reducing the steep growth of tail latency near saturation. Empirically, one expects the optimizer to shift work away from an overutilized device, even if that increases per-request service time, because the reduced queueing can improve high-percentile latency.

A key discussion point is the interaction between filtering and refinement. When attribute filters are selective, performing filtering on CPU before GPU refinement can reduce transfer volume and GPU work [63]. However, if filtering requires accessing many metadata fields and causes cache misses, CPU filtering can be expensive and can itself become the bottleneck. In such cases, pushing certain

filters to GPU, especially those expressible as simple numeric comparisons on packed arrays, can be beneficial. The engine supports this by allowing a split filtering operator where simple predicates run on GPU and complex predicates run on CPU, with a placement decision that minimizes crossings. The cost model must estimate the selectivity of each predicate subset and the cost of packing attributes [64]. If packing is amortized and the same attributes are used across many queries, the packing cost per request can be low, making GPU filtering more attractive.

Another discussion point is index residency. Keeping embeddings and index structures on GPU reduces transfer costs and allows fast scanning, but it requires significant GPU memory. If the catalog is large, full residency may be infeasible, requiring either compression, sharding, or CPU-based indices [65]. The hybrid engine can support a split approach in which coarse quantization structures and codebooks reside on GPU while inverted lists reside on CPU, or vice versa. Such split designs reduce memory pressure but introduce more transfers and complicate kernels. The cost-based placement must therefore incorporate memory capacity constraints and must evaluate alternative index layouts as part of the plan space [66]. In practice, memory constraints can dominate placement decisions, making it important that the optimizer reasons about memory explicitly rather than assuming ideal residency.

The evaluation also considers sensitivity to embedding dimension d and candidate set size K. As d increases, dot product refinement becomes more compute-heavy, making GPU acceleration more valuable. As K increases, candidate generation and scanning become heavier, potentially shifting bottlenecks [67]. The optimizer should adapt by placing refinement on GPU for large d and by placing candidate generation on the device that best matches the index type and residency. For small K, CPU execution may be preferred because overheads dominate. For large K, GPU may be preferred if the scan can be expressed with regular memory access. These regime changes are central to the motivation for cost-based placement, because a single fixed strategy cannot cover all combinations of d, K, selectivity, and load [68].

Limitations arise from modeling approximations. The cost model uses simplified representations of compute and memory costs and approximates queueing and overlap effects. Real systems exhibit nonlinearities such as cache effects, NUMA behavior on multi-socket CPUs, and interference from colocated workloads. Additionally, selectivity estimates can be wrong, especially when query intent shifts suddenly [69]. The optimizer mitigates this by using conservative estimates and by re-optimizing when telemetry indicates mismatch, but it cannot guarantee optimality. Another limitation is that some operators may not have efficient implementations on both devices, restricting placement. The benefits of the approach therefore depend on engineering investment in physical operator coverage and on careful kernel design for GPU operators.

8. Conclusion

This paper presented a hybrid CPU–GPU execution engine for similarity search in e-commerce product retrieval with cost-based operator placement [70]. The engine models retrieval as an operator graph that combines approximate candidate generation, structured filtering, exact refinement, and ranking computations, and it selects device placement and data residency using a calibrated cost model. The runtime supports asynchronous scheduling, explicit transfers, and micro-batching to balance GPU utilization with tail-latency constraints. The cost model accounts for compute and memory behavior, interconnect transfers, and utilization-driven queueing effects, enabling the optimizer to choose mixed placement plans that reduce unnecessary crossings while accelerating dense numeric kernels on the GPU.

The approach is designed for practical variability in e-commerce workloads, where selectivity, query mix, and traffic levels change over time [71]. By incorporating telemetry-driven calibration and by allowing periodic re-optimization, the engine can adapt placement without requiring constant manual retuning. The discussion highlighted the importance of modeling selectivity and memory residency, and it emphasized that tail latency is often determined by contention and overheads rather than by raw kernel throughput. Remaining challenges include improving model fidelity for interference and nonlinear cache effects, expanding operator coverage across devices, and integrating index-layout choices more deeply into the optimization space. Within these constraints, cost-based operator placement provides a structured way to exploit heterogeneous hardware for similarity search while respecting the operational requirements of e-commerce retrieval services [72].

References

[1] Y. Li, Z. Zhou, B. Sedlak, and S. Dustdar, "Mgg-ad: Multi-granularity graph-based anomaly detection in iot systems," in *2025 IEEE International Conference on Web Services (ICWS)*, pp. 1–10, IEEE, 7 2025.

[2] R. S. Chowhan, *Evolution and Paradigm Shift in Distributed System Architecture*. IntechOpen, 4 2019.

[3] H. Mansouri and A.-S. K. Pathan, *SSCC - A Resilient Hierarchical Checkpointing Algorithm for Distributed Systems Running on Cluster Federation.*, pp. 99–110. Germany: Springer Singapore, 4 2020.

[4] T. Peroutka, I. Murturi, P. K. Donta, and S. Dustdar, "A graph-based approach to human activity recognition," in *2024 IEEE 6th International Conference on Cognitive Machine Intelligence (CogMI)*, pp. 117–126, IEEE, 10 2024.

[5] R. Malik, S. Kim, X. Jin, C. Ramachandran, J. Han, I. Gupta, and K. Nahrstedt, "Mlr-index: An index structure for fast and scalable similarity search in high dimensions," in *International Conference on Scientific and Statistical Database Management*, pp. 167–184, Springer, 2009.

[6] S. Joshi, S. Ameta, and G. Lavania, "Balanced load in distributed system with nosql middleware," *Journal of emerging technologies and innovative research*, 5 2019.

[7] W. B. Daszczuk, "Modeling and verification of asynchronous systems using timed integrated model of distributed systems.," *Sensors (Basel, Switzerland)*, vol. 22, pp. 1157–1157, 2 2022.

[8] L. Nenov, "Reinforcement learning for key management in distributed systems," in *2024 32nd National Conference with International Participation (TELECOM)*, pp. 1–5, IEEE, 11 2024.

[9] V. Goronjic and S. Nastic, "Miso: A crdt-based middleware for stateful objects in the serverless edge-cloud continuum," in *2024 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 55–65, IEEE, 9 2024.

[10] S. K. Shukla and P. K. Sharan, "Logical clock implementation in the distributed system," *International Journal of Advance Research, Ideas and Innovations in Technology*, vol. 4, pp. 2214–2219, 4 2018.

[11] H. E. Fazazi, M. Elgarej, M. Qbadou, and K. Mansouri, "Design of an adaptive e-learning system based on multi-agent approach and reinforcement learning," *Engineering, Technology & Applied Science Research*, vol. 11, pp. 6637–6644, 2 2021.

[12] U. Matter, *Distributed Systems*, pp. 81–96. Chapman and Hall/CRC, 6 2023.

[13] N. V. Patel, "Adaptive experimentation in marketplaces: Balancing exploration and revenue optimization," *Frontiers in Health Informatics*, vol. 11, pp. 760–786, 2022.

[14] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle, "Secrets revealed in container images: An internet-wide study on occurrence and impact," in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pp. 797–811, ACM, 7 2023.

[15] F. Kargl, R. W. van der Heijden, B. Erb, and C. Bösch, *Privacy in Mobile Sensing*, pp. 13–23. Springer International Publishing, 7 2022.

[16] Y. Zhang, "Automatic failure diagnosis for distributed systems," 3 2021.

[17] Y. Mnaoui, A. Najoua, and H. Ouajji, *Automatic English Proficiency Assessment for Air Traffic Controllers in Emergencies*, pp. 996–1002. Springer International Publishing, 2 2022.

[18] I. Čilić, A. Lackinger, P. A. Frangoudis, I. P. Žarko, A. Furutanpey, I. Murturi, and S. Dustdar, "Reactive orchestration for hierarchical federated learning under a communication cost budget," in *2025 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pp. 1–7, IEEE, 5 2025.

[19] "Ieee distributed systems online," 3 2023.

[20] M. Qiu and S.-Y. Kung, "Special issue on 'smart computing and communication' in international journal of parallel, emergent and distributed systems," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35, pp. 217–218, 5 2020.

[21] V. Vijaykumar, R. Chandrasekar, and T. Srinivasan, "An obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs," in *2006 IEEE Conference on Cybernetics and Intelligent Systems*, pp. 1–6, IEEE, 2006.

[22] M. Zaki, "Balancing privacy, precision andperformance in distributed systems," 1 2020.

[23] C. Marcelino and S. Nastic, "Truffle: Efficient data passing for data-intensive serverless workflows in the edge-cloud continuum," in *2024 IEEE/ACM 17th International Conference on Utility and Cloud Computing (UCC)*, pp. 53–62, IEEE, 12 2024.

[24] S. Sadiq and S. R. M. Zeebaree, "Distributed systems for machine learning in cloud computing: A review of scalable and efficient training and inference," *Indonesian Journal of Computer Science*, vol. 13, 4 2024.

[25] T. Nowak, U. Schmid, and K. Winkler, "Topological characterization of consensus in distributed systems," *Journal of the ACM*, vol. 71, pp. 1–48, 11 2024.

[26] M. F. Rehme, S. Zimmer, and D. Pflüger, *Hierarchical Extended B-splines for Approximations on Sparse Grids*, pp. 187–203. Germany: Springer International Publishing, 10 2021.

[27] Y. Qian, S. Ying, and B. Wang, "Smc - anomaly detection in distributed systems via variational autoencoders," in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 2822–2829, IEEE, 10 2020.

[28] R. Lichtenthäler and G. Wirtz, "An experimental validation of architectural measures for cloud-native quality evaluations," in *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, pp. 374–384, IEEE, 7 2025.

[29] N. V. Patel, "Selection bias in two-sided e-commerce marketplaces: A framework for propensity score matching implementation," *Journal of Business Intelligence Systems and Computational Social Science Applications*, vol. 11, no. 5, pp. 1–20, 2021.

[30] R. Achar, P. Dawn, and C. V. Lopes, "Gotcha: An interactive debugger for got-based distributed systems," 1 2019.

[31] K. Vinnakota and M. Kolla, "Creating effective alerts for monitoring distributed systems," *International Journal of Computer Trends and Technology*, vol. 73, pp. 172–178, 5 2025.

[32] *Scalable Distributed Systems*, pp. 2290–2290. Springer New York, 6 2018.

[33] E. D. Caro, N. Belletti, F. Poltronieri, M. Tortonesi, and C. Stefanelli, "Beyond timegraph: A comparative analysis of temporal generators for evolving network graphs," in *2025 21st International Conference on Network and Service Management (CNSM)*, pp. 1–6, IEEE, 10 2025.

[34] K. W. HamaAli and S. R. M. Zeebaree, "Resources allocation for distributed systems: A review," 4 2021.

[35] F. Kargl, B. Erb, and C. Bösch, *Defining Privacy*, pp. 461–463. Springer International Publishing, 7 2022.

[36] C. L. Abad, E. Ortiz-Holguin, and E. F. Boza, "Sigcse - have we reached consensus? an analysis of distributed systems syllabi," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pp. 1082–1088, ACM, 3 2021.

[37] R. Chandrasekar and T. Srinivasan, "An improved probabilistic ant based clustering for distributed databases," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 2701–2706, 2007.

[38] F. Lai, P. Zhang, R. Cheng, and P. Xu, "Distributed systems anomaly detection based on log," in *2021 International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, vol. 14, pp. 72–79, IEEE, 12 2021.

[39] W. Wu, Z. Shen, and D. Wang, "A running pattern recognition algorithm for publish/subscribe distributed systems," *DEStech Transactions on Materials Science and Engineering*, 4 2021.

[40] N. S. Chatharajupalli, R. Rotta, R. Karnapke, and J. Nolte, "Probing considered harmful: Leveraging rssi for link quality prediction," in *2025 IEEE 50th Conference on Local Computer Networks (LCN)*, pp. 1–9, IEEE, 10 2025.

[41] M. Zakarya, L. Gillam, K. Salah, O. F. Rana, S. Tirunagari, and R. BUYYA, "Colocateme: Aggregation-based, energy, performance and cost aware vm placement and consolidation in heterogeneous iaas clouds," 6 2021.

[42] D. D. Gaudio, B. Ariguib, A. Bartenbach, and G. Solakis, "A live context model for semantic reasoning in iot applications," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 322–327, IEEE, 3 2022.

[43] N. Benmoussa, M. F. Amr, S. Ahriz, K. Mansouri, and E. Illoussamen, "Outlining a model of an intelligent decision support system based on multi agents," *Zenodo (CERN European Organization for Nuclear Research)*, 6 2018.

[44] W. Smith, T. Moyer, and C. Munson, *TaPP - Curator: Provenance Management for Modern Distributed Systems*. 6 2018.

[45] T. Srinivasan, R. Chandrasekar, V. Vijaykumar, V. Mahadevan, A. Meyyappan, and A. Manikandan, "Localized tree change multicast protocol for mobile ad hoc networks," in *2006 International Conference on Wireless and Mobile Communications (ICWMC'06)*, pp. 44–44, IEEE, 2006.

[46] G. Hu, X. Yao, and Y. Yu, "Collie: Federated query processing via reinforcement learning," in *2025 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 1284–1291, IEEE, 10 2025.

[47] F. A. Silva, I. Fé, C. Brito, G. Araújo, L. Feitosa, E. Choi, D. Min, and T. A. Nguyen, "Supporting availability evaluation of a smart building monitoring system aided by fog computing," *Electronics Letters*, vol. 58, pp. 471–473, 4 2022.

[48] D. Basin, M. Gras, S. Krstic, and J. Schneider, *RV - Scalable Online Monitoring of Distributed Systems*, pp. 197–220. Germany: Springer International Publishing, 10 2020.

[49] S. Gorai, *Decentralization without Blockchains*, pp. 24–32. Productivity Press, 8 2024.

[50] G. B. King, S. McCarthy, P. Pattabhiraman, J. Luciani, and M. Fleming, "Fallout: Distributed systems testing as a service," 1 2021.

[51] S. Gregersen, "Aneris: A mechanised logic for modular reasoning about distributed systems," 3 2021.

[52] C. Herbert, V. Marschin, B. Erb, D. Meißner, M. Aufheimer, and C. Bösch, "Are you willing to self-disclose for science? effects of privacy awareness and trust in privacy on self-disclosure of personal and health data in online scientific studies-an experimental study.," *Frontiers in big data*, vol. 4, pp. 763196–, 12 2021.

[53] R. Chandrasekar, R. Suresh, and S. Ponnambalam, "Evaluating an obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs," in *2006 International Conference on Advanced Computing and Communications*, pp. 628–629, IEEE, 2006.

[54] W. B. Daszczuk, *Fairness in Distributed Systems Verification*, pp. 139–159. Germany: Springer International Publishing, 3 2019.

[55] S. Devismes, "Versatility and efficiency in self-stabilizing distributed systems," 12 2020.

[56] H. Shukur, S. Zeebaree, R. Zebari, O. Ahmed, L. Haji, and D. Abdulqader, "Cache coherence protocols in distributed systems," *Journal of Applied Science and Technology Trends*, vol. 1, pp. 92–97, 6 2020.

[57] I. Schagaev and S. Farrell, *Distributed Systems: Resilience, Desperation*, pp. 343–368. Springer International Publishing, 7 2024.

[58] B. K. Ozkan, R. Majumdar, F. Niksic, M. T. Befrouei, and G. Weissenbacher, "Randomized testing of distributed systems with probabilistic guarantees," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 160–28, 10 2018.

[59] G. Heinrich and I. Frosio, "Metaoptimization on a distributed system for deep reinforcement learning," 2 2019.

[60] A. S. Hosseini and S. Staab, "Emotional framing in the spreading of false and true claims," in *Proceedings of the 15th ACM Web Science Conference 2023*, pp. 96–106, ACM, 4 2023.

[61] N. V. Patel, "Estimating heterogeneous treatment effects of driver incentives in ride-hailing platforms," *Northern Reviews on Algorithmic Research, Theoretical Computation, and Complexity*, vol. 9, no. 11, pp. 16–43, 2024.

[62] A. Hermann, N. Trkulja, P. Wachter, B. Erb, and F. Kargl, "Quantification methods for trust in cooperative driving," in *2025 IEEE Vehicular Networking Conference (VNC)*, pp. 1–8, IEEE, 6 2025.

[63] X. Fu, "Issta - towards scalable defense of information flow security for distributed systems," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 438–442, ACM, 7 2019.

[64] A.-M. Căilean, C. Beguni, and M. Dimian, "A novel approach for improved precision ranging based on vehicle-to-vehicle visible light communications: Concept and experimental demonstration," in *2025 International Conference on Applied Electronics (AE)*, pp. 1–4, IEEE, 9 2025.

[65] J. W. Baker, B. Bradley, and P. J. Stafford, *Spatially Distributed Systems*, pp. 447–482. Cambridge University Press, 10 2021.

[66] A. Furutanpey, P. A. Frangoudis, P. Szabo, and S. Dustdar, "Adversarial robustness of bottleneck injected deep neural networks for task-oriented communication," in *2025 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pp. 1–6, IEEE, 5 2025.

[67] K. Orynbekova, S. Kadyrov, A. Bogdanchikov, and S. Oktamov, "A novel recommender system for adapting single machine problems to distributed systems within mapreduce," *Bulletin of Electrical Engineering and Informatics*, vol. 14, pp. 687–695, 2 2025.

[68] H. Bouyghf, S. Abi, B. Benhala, and A. Raihani, "A comparison of metaheuristic techniques for rf passive components optimization," 11 2018.

[69] R. Malik, C. Ramachandran, I. Gupta, and K. Nahrstedt, "Samera: a scalable and memory-efficient feature extraction algorithm for short 3d video segments.," in *IMMERSCOM*, p. 18, 2009.

[70] M. ter Beek, B. Re, M. Viroli, R. Anane, and R. Bahsoon, "Session details: Distributed systems: Ccs track," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ACM, 4 2018.

[71] M. Senn and C. Cachin, "Asymmetric failure assumptions for reliable distributed systems," in *Proceedings of the 12th Workshop on Principles and Practice of Consistency for Distributed Data*, pp. 8–14, ACM, 3 2025.

[72] L. Grote, I. Kunze, C. Sander, and K. Wehrle, "Instant messaging meets video conferencing: Studying the performance of im video calls," in *2023 7th Network Traffic Measurement and Analysis Conference (TMA)*, pp. 1–10, IEEE, 6 2023.