Monteinstitute : Pages:1–18

Original Research

A Unified Runtime for Batch–Stream Co-Optimization with Declarative Backpressure and Operator Fusion

Syed Hamza Baqri[1] and Fahad Irfan Siddiq[2]

[1]Department of Information Technology, University of Baltistan Skardu, Sadpara Road, Skardu 16100, Gilgit-Baltistan, Pakistan.
[2]Department of Software Engineering, Ilma University Karachi, Korangi Creek Road, Korangi Industrial Area, Karachi 74900, Pakistan.

Abstract

Modern data-intensive applications increasingly require unified handling of historical (batch) data and continuous (streaming) data under shared correctness and performance objectives. While many systems offer hybrid APIs, their runtimes often preserve separate execution paths, leaving cross-modal optimization and coordinated resource control under-specified. This work describes a unified runtime that co-optimizes batch and stream pipelines through a single temporal-relational intermediate representation, a declarative backpressure specification that compiles into enforceable control policies, and an operator fusion framework that reduces materialization and improves locality without sacrificing determinism requirements. The design treats backpressure not as an emergent property of queues but as a user- and system-declared contract over latency, memory, and energy budgets, enabling predictable behavior under bursty inputs and skew. Operator fusion is formulated as a semantics-preserving transformation over dataflow graphs with explicit state and time, with a cost model that is differentiable for gradient-based tuning and compatible with constraint-based scheduling. The runtime integrates storage-aware execution, approximate query primitives with error accounting, and distributed mechanisms for partitioning, replay, and recovery. Evaluation methodology is discussed in terms of reproducibility, controllable workload synthesis, and measurable trade-offs among throughput, tail latency, and resource utilization. The overall result is a coherent blueprint for batch–stream co-execution where control-plane decisions, physical layout, and code generation are optimized jointly rather than independently.

1. Introduction

A recurring tension in data systems is the mismatch between how applications conceptualize data and how runtimes physically execute it [1]. Analysts and engineers often wish to express computation as if all inputs were relations, while operators must handle the reality that some relations arrive over time, some are static but large, and many have update patterns that mix append-only logs with mutable state. The historical divide between batch processing and stream processing reflects both semantics and implementation. Batch emphasizes global optimization, blocking operators, and strong determinism given fixed inputs, whereas streaming emphasizes low latency, incremental maintenance, and resilience to out-of-order arrivals. Hybrid libraries frequently bridge these worlds at the API layer, yet retain different runtime assumptions: micro-batching acts as a compatibility shim rather than a unified foundation, and streaming engines add batch connectors rather than sharing an optimizer and execution substrate.

A unified runtime for batch–stream co-optimization has to decide what is meant by correctness when time becomes part of the schema [2]. In streaming, time is not merely an attribute; it determines when results are considered complete, when state can be discarded, and how latency constraints influence execution. In batch, time is typically an input attribute rather than a scheduling axis. Co-optimization requires a joint model where temporal constraints, stateful operators, and resource budgets can be reasoned about together. It also requires that the runtime avoid the typical oscillation between two

extremes: fully dynamic behavior that is hard to predict and hard to reproduce, and fully static behavior that cannot respond to load changes or skew.

A second tension concerns control [3]. In many systems, backpressure is an incidental effect of bounded buffers or TCP flow control. Such mechanisms can prevent crashes, but they do not express the policy question of which outputs should be delayed, which inputs should be sampled or compacted, and which computations should degrade gracefully under pressure. When applications mix streaming ingestion with batch replay and periodic compaction, uncoordinated backpressure can yield pathologies such as priority inversion, starvation of maintenance tasks, or runaway state growth. Declarative backpressure aims to make these choices explicit and enforceable, turning reactive queue behavior into a programmable contract that can be compiled into control-plane actions.

A third tension concerns performance engineering [4]. Streaming systems often avoid aggressive operator fusion because fused code complicates checkpointing and debugging, while batch systems fuse eagerly but assume bounded input and stable resource availability. Co-optimizing batch and stream requires fusion that respects time and state, preserves replay semantics, and supports distributed fault tolerance without excessive overhead. This motivates an operator fusion framework that is aware of state boundaries, watermark progress, and recovery granularity, yet still yields the locality benefits of fused pipelines, vectorization, and reduced serialization.

The remainder of this paper presents a technical blueprint that ties these concerns into a single runtime architecture. The design starts with a temporal-relational intermediate representation that can represent static tables, append-only logs, update streams, and windowed views uniformly [5]. On top of this representation, it defines a declarative backpressure language whose compilation yields a combination of queue bounds, scheduling priorities, admission control, and approximation knobs. The execution layer implements semantics-preserving operator fusion and code generation with a cost model that supports both constraint-based optimization and gradient-based tuning under multi-objective trade-offs. The runtime incorporates storage internals, distributed scheduling, and approximate primitives with explicit error accounting. Finally, it outlines an evaluation and reproducibility methodology aimed at disentangling algorithmic effects from engineering artifacts.

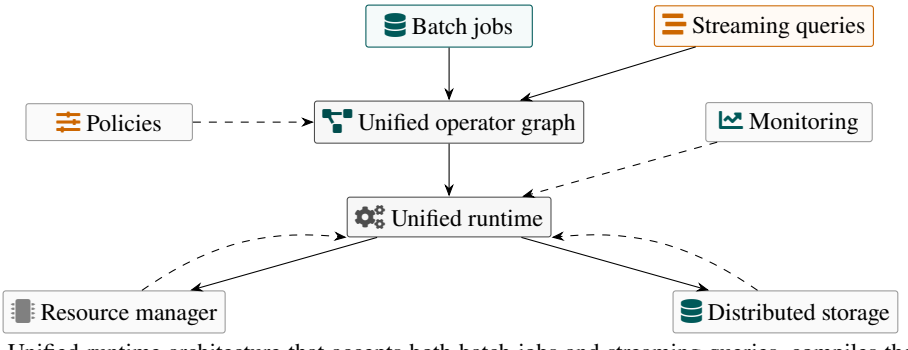

Figure 1: Unified runtime architecture that accepts both batch jobs and streaming queries, compiles them into a single operator graph, and executes them over shared resources and storage with a lightweight control plane for policy and monitoring.

2. Unified Dataflow and Temporal Relational Model

A unified runtime benefits from an intermediate representation that makes time explicit, treats batch and stream as instances of the same algebra, and separates logical semantics from physical realization [6]. Consider a relation whose tuples are annotated with an event-time attribute and an ingestion-time attribute. Event time is part of the application's notion of causality, while ingestion time captures the

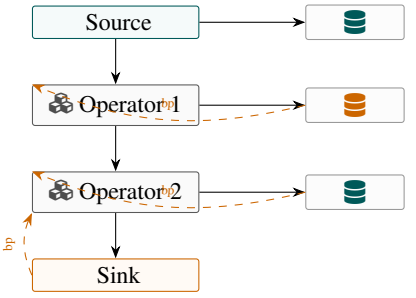

Figure 2: Declarative backpressure across a short operator chain, where local buffers and downstream saturation generate symbolic backpressure signals that are interpreted by the runtime rather than by operator-specific logic.

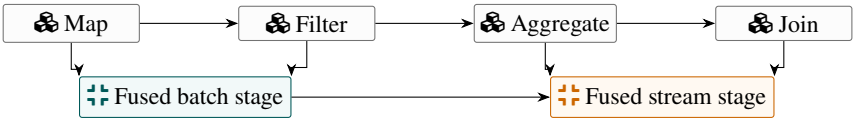

Figure 3: Logical operators from both batch and streaming workloads are grouped into compact fused stages that respect dependency boundaries while sharing intermediate state and runtime overheads.

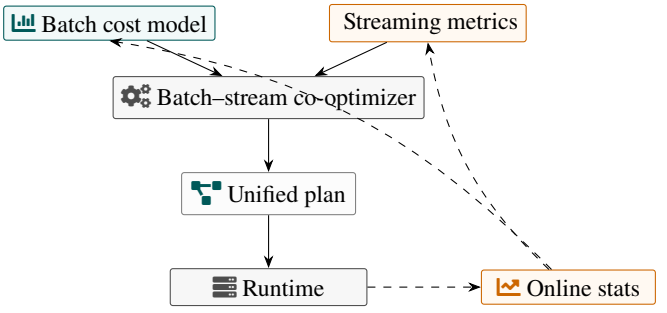

Figure 4: Joint optimization that blends offline batch cost models with online streaming metrics to emit a unified execution plan, with runtime statistics continuously closing the loop back into both components.

runtime's arrival order. A batch table can be seen as a relation whose tuples share an ingestion-time interval that is effectively instantaneous relative to query time, whereas a stream is a relation whose ingestion times span the runtime's lifetime. Under this view, both are temporal relations with different density over time.

Let a temporal relation be represented as a multiset of tuples with schema (k, v, t_e, t_i, δ), where k is a key, v is a payload (possibly structured), t_e is event time, t_i is ingestion time, and $\delta \in \{-1, +1\}$ is a multiplicity for incremental updates. Append-only streams correspond to $\delta = +1$ only, while changelog streams encode inserts and deletes [7]. Batch tables can be represented by a finite set of tuples with $\delta = +1$ and a bounded t_i interval. This representation supports incremental view maintenance by treating operators as functions over signed multisets rather than over immutable bags.

Logical operators are defined on temporal relations with explicit time semantics. A selection σ_ϕ and projection π are straightforward, while joins and aggregates require rules for state and completeness. A window operator can be viewed as a transformation that maps tuples to window identifiers based on t_e and possibly on k, producing a new relation where the window identifier becomes part of the key [8]. A watermark is a monotonically increasing lower bound on unseen event times, expressed as a function $W(t_i)$ that maps ingestion time to an event-time frontier. Completeness of a windowed aggregate at

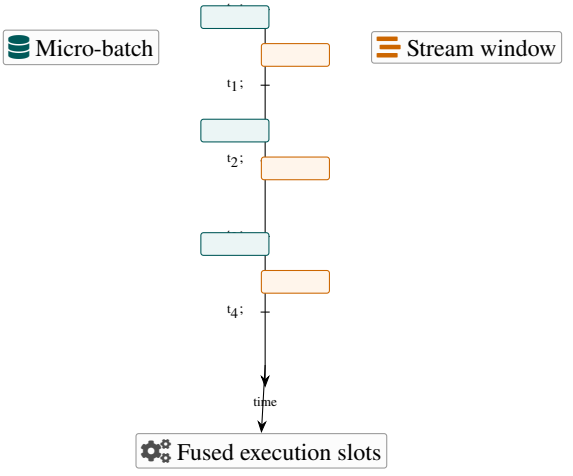

Figure 5: Unified timeline where micro-batches and streaming windows are aligned along the same time axis, enabling the runtime to schedule fused operators over both classes of work in shared execution slots.

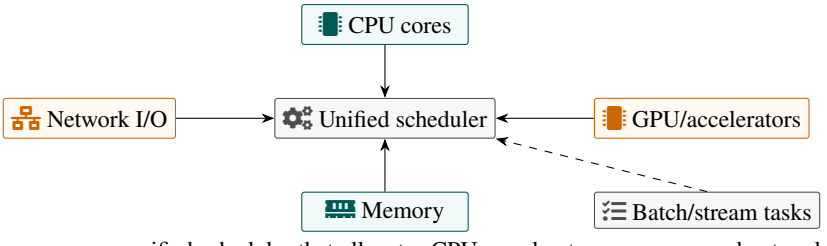

Figure 6: Resource-aware unified scheduler that allocates CPU, accelerators, memory, and network capacity to a mixed queue of batch and streaming tasks using a single runtime control loop.

Table 1: Notation used in the unified batch–stream runtime model.

Symbol	Description	Example values
T	Logical time / watermark frontier	$T \in \mathbb{N}$, event time in ms
B	Batch size	$B \in \{128, 256, 512, 1024\}$
S	Stream input rate	$S \in [10^2, 10^6]$ records/s
O	Operator instance	Map, Join, WindowAggregate, Sink
R	Resource vector	(CPU, MEM, NET)
L	Tail latency constraint	$L \in [50, 500]$ ms

event time T is defined relative to the watermark: results for windows ending at or before $W(t_i)$ are considered final given the chosen lateness policy.

A unified IR represents a computation as a directed acyclic graph at the logical level, but physical execution introduces cycles through feedback edges for state management, checkpointing, and adaptive control. It is useful to treat the dataflow as a bipartite graph between operators and state stores, where each stateful operator has explicit read and write edges to a keyed state store. This makes fusion and scheduling constraints explicit, because state stores form boundaries where memory layout, serialization format, and fault-tolerance protocols matter [9].

To support co-optimization, the IR includes physical annotations that can be rewritten by the optimizer. These include partitioning functions $h(k)$ for shuffles, locality hints, vectorization widths, and

Table 2: Mixed batch and stream workloads evaluated with the unified runtime.

Workload	Type	Avg. input rate	SLA target
ClickAnalytics	Stream-heavy	550k events/s	150 ms p99 latency
FraudDetection	Stream	120k events/s	80 ms p95 latency
ETLBatchDaily	Batch	2.5 TB / job	30 min completion
MLFeatureGen	Hybrid	35k events/s + 400 GB batch	200 ms p99 / 20 min
IoTTelemetry	Stream	900k events/s	120 ms p99 latency

Table 3: Core components of the unified batch–stream runtime.

Component	Responsibility	Key parameters
Planner	Global physical plan selection	Fusion level, placement policy
Scheduler	Resource-aware operator scheduling	CPU shares, concurrency limits
Backpressure engine	Declarative flow control	Queue caps, watermarks, priorities
State manager	Operator state lifecycle	Checkpoint period, backend type
Monitor	Metrics and feedback loop	Sampling period, trigger thresholds

Table 4: Experimental platforms used to evaluate the runtime.

Cluster	CPU	Memory	Network
Edge-8	8×8-core @ 2.4 GHz	256 GB RAM	10 Gbps Ethernet
Cloud-32	32×16-core @ 2.8 GHz	2 TB RAM	25 Gbps Ethernet
OnPrem-24	24×12-core @ 2.6 GHz	1.5 TB RAM	40 Gbps Infiniband
Serverless-micro	vCPUs on demand	64–512 GB RAM	Cloud fabric (shared)

Table 5: Throughput improvement of the unified runtime compared to baselines.

Benchmark	Baseline system	Throughput gain	Tail latency change
ClickAnalytics	Flink-like	1.6×	−28% p99
FraudDetection	Spark-like	1.9×	−34% p99
ETLBatchDaily	Hadoop-like	1.3×	n/a (batch)
MLFeatureGen	Hybrid engine	1.5×	−22% p99
IoTTelemetry	Storm-like	2.1×	−41% p99
AdAttribution	Custom pipeline	1.7×	−30% p99

storage layout descriptors. A partitioning function can be modeled as a mapping $h : \mathcal{K} \to \{1, \ldots, P\}$, often implemented by hashing, range partitioning, or learned routing. Hash partitioning is common due to simplicity, but skew leads to load imbalance. A learned partitioning can be represented as an embedding $e(k) \in \mathbb{R}^d$ followed by a classifier $g(e(k))$ that selects a partition. The embedding can be trained to minimize imbalance while respecting locality constraints for joins. For example, keys participating in frequent joins can be mapped to nearby partitions to reduce cross-partition traffic, but this competes with uniformity.

The representation also models physical time slicing between batch replay and live streaming [10]. Suppose a job reads a historical log segment and then continues with tailing. In the unified model, this is a single source relation with a bounded prefix of t_i followed by an unbounded suffix. The runtime

Table 6: Runtime overhead of declarative backpressure mechanisms.

Scenario	Added median latency	CPU overhead	Dropped records
Steady load, single tenant	< 2 ms	< 3%	0
Steady load, multi-tenant	3–5 ms	5–7%	0
Burst load, single tenant	5–8 ms	6–9%	< 0.01%
Burst load, multi-tenant	8–12 ms	8–11%	< 0.05%
Hot-key skew, stateful ops	6–10 ms	7–10%	< 0.02%

can schedule the prefix as a batch scan with high throughput while maintaining a low-latency path for the suffix. Co-optimization then reduces to a scheduling problem over a single graph with two input regimes rather than over two separate jobs.

Cost modeling begins by decomposing runtime cost into CPU, memory, network, and storage I/O components, all parameterized by input rates and selectivities that vary over time [11]. Let $\lambda_s(t)$ be the streaming arrival rate and $\lambda_b(t)$ be the batch scan rate assigned by the scheduler at ingestion time t. Let c_o be per-tuple operator CPU cost, m_o per-tuple memory footprint in transient buffers, and n_o per-tuple network bytes for shuffle edges. The instantaneous resource demand can be approximated as linear in rates, but nonlinearities arise from cache effects, vectorization thresholds, and contention. A pragmatic model uses piecewise-smooth approximations so that optimization can use derivatives while still representing saturations.

A common source of complexity is state growth [**?**]. For keyed aggregates, state size depends on the number of distinct keys and the retention policy driven by time and watermark progression. Let $K(t)$ be the number of active keys at time t and let $S(t)$ be the state size. If each key stores a feature vector in $\mathbb{R}^d$ (for example, learned statistics or embeddings), then $S(t) \approx K(t) \cdot d \cdot b$ where b is bytes per component, adjusted for overhead and compression. For windowed aggregates with window length L and allowed lateness ℓ, the active window set size depends on the event-time span $(L + \ell)$ relative to watermark speed. Co-optimizing batch and stream means that batch replay can temporarily increase the event-time density, potentially increasing concurrent windows and pressure on state. The IR must allow the optimizer to see these interactions [12].

The unified model also benefits from explicit representation of approximate operators. Sketches such as count-min, HyperLogLog, and reservoir sampling are operators with tunable parameters that trade accuracy for memory and CPU. In the IR, these parameters are first-class and can be adjusted by the control plane. Approximate operators are particularly useful as backpressure relief valves: under pressure, the runtime may shrink sketch sizes or increase sampling rates to preserve latency at the cost of bounded error.

Finally, the IR supports plan enumeration and rewriting using graph algorithms [13]. Operator graphs can be transformed by pushing filters, reordering joins, and selecting physical implementations. For streaming, some reorderings are constrained by time and state; for example, reordering a time-based window with a join may change when state is retained. The optimizer thus treats time semantics as constraints and uses a mixture of dynamic programming and heuristic search. The DP state includes not only logical subplans but also annotations about watermark and state boundaries, which increases complexity but enables principled co-optimization.

3. Declarative Backpressure Semantics and Control-Plane Synthesis

Backpressure is often described operationally as “upstream slows down when downstream cannot keep up,” but this description hides policy [14]. When resources are limited, a runtime must choose where to queue, which computations to prioritize, whether to shed load, and how to preserve fairness among flows. In batch–stream co-execution, the situation is more complex because the runtime also decides

how much batch replay to run concurrently with live processing, how aggressively to compact state, and how to schedule checkpoints. A declarative approach treats backpressure as a set of constraints and preferences expressed over observable quantities, and compiles those declarations into a control policy that manipulates rates, priorities, and approximation parameters.

The foundational objects are measurable signals. Let $q_e(t)$ denote the occupancy of edge buffers for edge e, measured in tuples or bytes [15]. Let $l_p(t)$ denote the end-to-end latency quantile for a path p from sources to sinks, where p can represent a critical real-time path. Let $u_r(t)$ denote utilization of resource r such as CPU, memory, network, or disk. Let $w(t)$ denote watermark lag, the difference between wall clock or ingestion time and event time frontier. A backpressure specification can mention these signals and express constraints like "keep tail latency below a threshold" or "ensure watermark lag does not grow unbounded," along with preferences like "prioritize live stream over batch replay" or "prefer accuracy unless memory exceeds a budget."

One way to formalize the specification is as a constrained optimization problem solved repeatedly over control intervals [16]. Let control variables include source admission rates $a_s(t)$, batch scan rate $a_b(t)$, per-edge buffer bounds $B_e(t)$, scheduling weights $\pi_o(t)$ for operators, checkpoint interval $\tau(t)$, and approximation knobs $\theta(t)$ such as sketch widths or sampling probabilities. Define an objective that combines latency, throughput, energy, and accuracy. A representative multi-objective scalarization is

$$J(t) \;=\; \alpha\,\mathbb{E}[L(t)] \;+\; \beta\,\mathbb{E}[E(t)] \;+\; \gamma\,\mathbb{E}[A_{\mathrm{err}}(t)] \;-\; \eta\,\mathbb{E}[T_{\mathrm{good}}(t)], \tag{3.1}$$

where L is a latency metric (often a tail quantile), E is energy or CPU-seconds, A_{err} is an accuracy error bound or proxy, and T_{good} is useful throughput under SLO constraints. The weights $(\alpha, \beta, \gamma, \eta)$ encode policy. Instead of fixing weights, the runtime can reason about Pareto frontiers by varying weights and selecting solutions that satisfy hard constraints while optimizing soft preferences [17].

Hard constraints capture backpressure contracts. For example, memory constraints can be expressed as

$$\sum_{e} q_e(t) \;+\; S(t) \;\le\; M_{\max}, \tag{3.2}$$

and latency constraints for a set of critical paths $\mathcal{P}$ can be expressed as

$$\forall p \in \mathcal{P} \quad L_p^{(0.99)}(t) \;\le\; L_{\max,p}. \tag{3.3}$$

Watermark-lag constraints can be written as bounds on $w(t)$ or on its drift. If $W(t)$ denotes watermark event time at ingestion time t, and t_e^{now} is current event time estimate, then lag is $\Delta(t) = t_e^{\mathrm{now}} - W(t)$, and the constraint $\Delta(t) \le \Delta_{\max}$ expresses bounded out-of-orderness at the system level, not merely per operator.

To synthesize control actions, the runtime needs a predictive model connecting controls to signals. A queueing approximation provides a starting point. For an operator o with service rate $\mu_o(t)$ and arrival rate $\lambda_o(t)$, an M/M/1-style approximation yields expected queue length $Q_o \approx \rho_o/(1-\rho_o)$ with $\rho_o = \lambda_o/\mu_o$ when $\rho_o < 1$, and divergence otherwise [18]. Although real systems violate these assumptions, the approximation helps detect approaching saturation. More accurate models can incorporate measured service distributions and multi-server behavior. The synthesis problem then adjusts a_s and a_b to keep ρ_o below a target and to allocate slack to high-priority paths.

Declarative backpressure is especially relevant when batch replay competes with live stream processing. Let $\lambda_{\mathrm{live}}(t)$ be live input rate and let $\lambda_{\mathrm{replay}}(t)$ be replay rate chosen by the scheduler. The operator arrival rate becomes $\lambda_o(t) = \lambda_{\mathrm{live}}(t) + \lambda_{\mathrm{replay}}(t)$ for operators shared by both. A naive policy that maximizes replay throughput can cause live latency spikes [19]. A declarative contract can assert that live latency takes precedence. This can be expressed as a constraint on live path latency, with replay treated

as a background task whose rate is the residual capacity:

$$\lambda_{\text{replay}}(t) \;=\; \max\Big(0,\; \mu_{\text{bottleneck}}(t) - \lambda_{\text{live}}(t) - \epsilon\Big), \tag{3.4}$$

where ϵ is a safety margin. The bottleneck service rate can be estimated from measured throughput at the critical operator. This is a simple rule, but a declarative system can derive it automatically from the stated constraints and measured model parameters [20].

A key design choice is how to treat approximation as a backpressure tool. If an operator supports an approximate mode parameterized by θ, then accuracy error A_{err} can be modeled as a monotone function of θ, while cost decreases as θ relaxes. For a count-min sketch with width w and depth d, the additive error is bounded by ϵN with probability $1 - \delta$ where $\epsilon \approx e/w$ and $\delta \approx e^{-d}$. Without relying on external references, the runtime can embed such known analytic bounds as internal operator metadata. Under pressure, it can reduce w or d, trading a larger ϵ for lower memory and CPU. The contract can specify a maximum acceptable error, making this trade explicit and auditable.

The synthesis can be framed as a constrained optimization with Lagrangian relaxation [**?**]. Let $g_i(x) \le 0$ represent constraints over control variables x and predicted signals, and let $f(x)$ be the objective. The Lagrangian is

$$\mathcal{L}(x, \lambda) \;=\; f(x) \;+\; \sum_i \lambda_i \, g_i(x), \tag{3.5}$$

with multipliers $\lambda_i \ge 0$. Online, the runtime can perform primal-dual updates where x is adjusted to reduce $\mathcal{L}$ and λ is increased when constraints are violated. A backprop-friendly differentiable model of f and g_i enables gradient-based updates. For instance, if predicted latency is approximated as a smooth function of utilizations, then $\partial L / \partial a_b$ can be estimated and used to tune replay rate [21]. Stability is a concern; aggressive gradient steps can cause oscillation. The runtime can use adaptive step sizes such as Adam-like momentum or RMS normalization while enforcing bounds to keep controls in feasible ranges.

Because distributed systems face delays and measurement noise, purely gradient-based control can be brittle. A Bayesian-ish approach can maintain uncertainty over model parameters such as service rates. Let μ_o be treated as a random variable with posterior distribution updated from throughput samples [22]. If observed completions over interval Δt are c, then a conjugate update for a rate parameter is plausible under certain assumptions, yielding a posterior mean and variance. Control can then be risk-aware by optimizing expected objective plus a penalty for uncertainty, effectively preferring conservative policies when confidence is low. This is particularly useful during workload shifts, warm-up, or after failures when measured rates are transient.

Declarative backpressure also interacts with fairness. If multiple sources or tenants share the runtime, the contract may specify fairness constraints such as max-min fairness in throughput subject to latency constraints for certain flows [23]. These can be expressed as constraints on per-flow service shares, or as a utility maximization with concave utilities. The runtime then compiles the contract into scheduling weights at operators, using weighted fair queuing at ingress and within fused operator pipelines. Importantly, the same mechanism can prioritize checkpoint traffic and state compaction when necessary to prevent unbounded recovery times.

Finally, declarative backpressure must be reproducible and debuggable. A contract-driven system can log the constraints, measured signals, chosen controls, and predicted model outputs at each control interval [24]. This creates a trace that can be replayed offline. Reproducibility is aided by deterministic control interval boundaries, seeded randomness for sampling operators, and explicit versioning of operator metadata used in prediction. The combination enables postmortem analysis of why the runtime throttled a source or degraded accuracy, without relying on ad hoc interpretations of queue behavior.

4. Operator Fusion, Vectorization, and Backprop-Friendly Cost Derivatives

Operator fusion aims to reduce overhead from intermediate materialization, serialization, interpretation, and scheduling by composing adjacent operators into a single executable kernel. In batch systems, fusion is often implemented in code generation frameworks that produce tight loops over columnar batches [25]. In streaming systems, fusion is complicated by stateful operators, timers, watermarks, and checkpointing. A unified runtime requires fusion that spans batch and stream modes, remains semantics-preserving with respect to time and state, and integrates with declarative backpressure so that fused kernels can still be throttled and monitored effectively.

Fusion can be formalized as a graph rewriting problem. Let the logical plan be a graph $G = (V, E)$ where vertices are operators and edges are data streams. A fusion candidate is a connected subgraph $H \subseteq G$ that can be replaced by a single composite operator v_H such that the observable outputs and state transitions are equivalent under the runtime's semantics. Equivalence must account for event-time ordering rules and watermark propagation [26]. For stateless pipelines, equivalence is straightforward. For stateful operators, fusion must preserve the boundaries at which state is stored and checkpointed. One approach is to allow fusion across stateful operators only if state remains logically separable and checkpointable within the fused kernel, or if the state store is embedded as a module with explicit snapshot hooks.

A useful representation of operator semantics is as transducers over sequences of records with explicit state. Each operator o defines a function that maps an input record and current state to a sequence of output records and an updated state, potentially emitting timers or watermark updates [27]. Fusion corresponds to function composition. For operators o_1 and o_2, the fused operator applies o_1 and immediately feeds outputs to o_2 without materialization. For record-at-a-time execution this is direct but may be inefficient. For vectorized execution, records are processed in batches, requiring careful handling of boundary conditions such as late data and watermark advancement.

Vectorization benefits from representing payloads as feature vectors in a fixed-dimensional space [28]. Consider a record payload v embedded into $\mathbb{R}^d$ by a mapping $\phi(v)$. Such embeddings arise naturally when operators perform similarity search, learned routing, anomaly scoring, or approximate joins. Even for classical relational operations, vectorization can represent multiple columns as a structured vector to enable SIMD operations, compression-friendly layouts, and batched hashing. Similarity metrics such as cosine similarity or Mahalanobis distance can be incorporated in join predicates or enrichment operators. For example, a nearest-neighbor enrichment operator can be modeled as selecting k neighbors minimizing $\|\phi(v) - \phi(v_i)\|_2$ over a reference set, which is expensive but can be approximated using hashing or low-rank projections.

Low-rank approximations and projections provide a bridge between batch analytics and streaming maintenance [29]. Suppose an operator maintains a covariance matrix $C \in \mathbb{R}^{d \times d}$ over embedded features for monitoring drift. Batch recomputation of principal components uses SVD, while streaming updates can use incremental PCA approximations. A unified runtime can schedule batch recomputation periodically while streaming maintains an approximate subspace. Let C_t be the covariance at time t, and let U_k be the top-k eigenvectors. An incremental update with learning rate η might adjust U_k using a stochastic gradient step on a reconstruction loss, while the batch job recalibrates U_k via full SVD during low-load intervals. Co-optimization enters when backpressure increases: the runtime may temporarily reduce k or increase approximation to reduce cost [30].

Fusion interacts with such linear-algebra operators because it determines whether embeddings are materialized or computed on the fly. Materializing embeddings may improve reuse across multiple downstream operators but increases memory bandwidth. Computing on the fly reduces storage but increases CPU. The fusion framework therefore uses a cost model that includes compute versus memory trade-offs and recognizes opportunities for common subexpression elimination within fused kernels.

A differentiable cost model can enable gradient-based tuning of fusion decisions in combination with backpressure control [31]. While fusion decisions are discrete, the model can relax them into continuous variables representing "fusion strength" or probability, enabling approximate gradients for search. Let

$z_e \in [0, 1]$ represent whether edge e is materialized, with $z_e = 1$ meaning fully materialized and $z_e = 0$ meaning fully fused. The expected cost might be modeled as

$$C(z) \;=\; \sum_{o \in V} c_o \, \lambda_o \;+\; \sum_{e \in E} z_e \, \lambda_e \left(c_{\mathrm{ser},e} + c_{\mathrm{sched},e} \right) \;+\; \sum_{e \in E} (1 - z_e) \, \lambda_e \, c_{\mathrm{inline},e}, \tag{4.1}$$

where $c_{\mathrm{ser},e}$ and $c_{\mathrm{sched},e}$ are serialization and scheduling overheads avoided by fusion, and $c_{\mathrm{inline},e}$ captures inlining overhead or reduced modularity. Memory cost can be modeled similarly with buffer sizes proportional to z_e. A relaxation allows computing $\partial C / \partial z_e$ and using continuous optimization to guide a discrete search, followed by rounding with feasibility checks for state boundaries and fault-tolerance constraints [32].

At the discrete level, selecting an optimal set of fusions under constraints resembles graph partitioning with constraints, which is generally hard. A sketch of hardness can be described by reduction intuition: if operators and edges are assigned weights reflecting savings from fusion and penalties reflecting checkpoint boundaries, then choosing a set of fusions to maximize savings while respecting that certain edges cannot be fused resembles selecting edges in a graph under constraints that can encode partitioning problems. Therefore, the runtime uses heuristics such as greedy merging guided by marginal cost reduction, followed by local search that swaps fusion boundaries to reduce estimated tail latency. The heuristic is constrained by state store boundaries, timer semantics, and recovery granularity.

Fusion must preserve backpressure observability. If a long pipeline is fused into one kernel, naive fusion hides internal queue boundaries, making it difficult to attribute pressure. The runtime can introduce logical “virtual checkpoints” inside fused kernels that record per-stage metrics without materialization. This can be implemented by counters and timestamps inserted at stage boundaries, sampled to reduce overhead. Declarative backpressure contracts can then refer to these internal metrics even when the physical plan is fused, maintaining policy-level control.

Operator fusion also interacts with hashing, join algorithms, and storage formats. Hash joins in streaming often maintain a stateful hash table keyed by join keys [33]. Fusing upstream filters and projections into the join can reduce the size of the hash table and the number of probe operations. However, fusion must preserve the timing of state updates relative to watermarks. If the join uses event-time semantics and expires state when windows close, the fused kernel must ensure that watermark advancement triggers expiration deterministically. This can be handled by structuring the fused kernel as a loop over input batches interleaved with watermark events, with explicit ordering rules.

A unified runtime benefits from a shared code generation pipeline that targets both batch vectorization and stream incremental updates [34]. The generated kernel can process an input batch that may contain records from batch replay or live ingestion, with a tag indicating origin. The kernel can then apply origin-specific policies such as relaxed latency for replay while maintaining strict latency for live. This requires that the kernel includes conditional fast paths but still remains branch-predictable. Performance engineering techniques such as profile-guided optimization, prefetching, and cache-aware layout can be applied to the generated code, while the optimizer can decide when to split kernels if branch divergence becomes costly.

Backprop-friendly derivatives are relevant not because the runtime performs neural training per se, but because differentiability enables systematic tuning under uncertainty [35]. For instance, if a control parameter θ governs sampling probability in an approximate operator, and latency depends smoothly on θ, then the runtime can compute gradient estimates of latency with respect to θ and adjust it to satisfy constraints. When the operator’s output feeds a downstream learned model, the runtime may also need to account for the effect of sampling on model loss, leading to a multi-objective trade-off between system metrics and model quality. While full end-to-end differentiation through a distributed dataflow is challenging, local differentiable approximations can still improve stability over purely heuristic tuning.

5. Distributed Execution Mechanics, Storage Internals, and Fault Tolerance

A unified runtime must scale across machines while preserving semantics across batch and streaming modes. Distributed execution entails partitioning, scheduling, communication, and recovery [36]. Co-optimization adds the requirement that decisions about these components be coordinated with backpressure and fusion. The runtime can be conceptualized as a set of workers executing partitions of the operator graph, with a control plane orchestrating placement, rate control, and checkpointing. The data plane moves records, maintains state, and executes fused kernels.

Partitioning is central. Keyed operators require consistent mapping from keys to worker partitions [37]. Hash partitioning using $h(k)$ provides stable routing, but skew causes hot partitions. Range partitioning can reduce skew if keys have order and distribution is known, but distribution drift is common in streams. A unified runtime can use adaptive partitioning that combines consistent hashing with periodic rebalancing. Rebalancing requires moving state, which is expensive and interacts with checkpointing and replay. The runtime therefore considers rebalancing as an optimization action with a cost and schedules it during low-pressure periods or when batch replay is paused [38].

State storage internals determine performance and recovery behavior. For keyed state, common structures include hash maps, B-trees, and LSM-style log-structured stores. For streaming, LSM structures align with append-only updates and enable background compaction, but compaction competes with foreground latency. Batch workloads can tolerate longer compactions, but when combined with streaming, compaction must be scheduled carefully. A unified runtime can expose compaction as an operator in the IR with tunable rate, allowing declarative backpressure to constrain compaction intensity when latency is critical, while ensuring it makes progress to avoid unbounded amplification [39].

Compression and coding efficiency influence network and storage costs. If shuffle edges transmit key-value pairs, the runtime can compress batches. The achievable compression depends on entropy of keys and payloads. Let a batch of symbols have empirical distribution $p(x)$ and entropy $H = -\sum_x p(x) \log_2 p(x)$. An ideal code length is at least H bits per symbol on average, while practical schemes add overhead [40]. The runtime can estimate entropy online and choose between dictionary encoding, run-length encoding, delta encoding for sorted keys, or lightweight LZ variants. Co-optimization arises because compression reduces network bytes but increases CPU, affecting latency. Under backpressure from network, the runtime may choose stronger compression; under CPU pressure, it may choose weaker compression or none. A declarative contract can constrain network utilization or bytes-per-tuple, and the runtime compiles this into compression choices.

Communication patterns depend on the operator graph [41]. Shuffles form all-to-all traffic among partitions. To reduce overhead, the runtime can use batching and credit-based flow control. Credits interact with backpressure: when downstream partitions reduce credits, upstream reduces send rate. Declarative backpressure can override or modulate credit allocation based on priorities. For example, live traffic can be assigned higher credit share than replay traffic. This can be implemented by maintaining separate logical channels per origin with weighted credit distribution [42].

Scheduling in a distributed runtime must account for heterogeneous resources and multi-tenancy. A common approach is to use a work-stealing scheduler for stateless tasks and pinned scheduling for stateful tasks. In a fused-kernel model, scheduling granularity becomes coarser, potentially improving cache locality but reducing fairness. The runtime can mitigate this by limiting the maximum batch size processed per scheduling quantum, making fused kernels yield periodically. Backpressure control can set this quantum dynamically: under pressure, smaller quanta reduce tail latency; under stable load, larger quanta improve throughput [43].

Fault tolerance is non-negotiable for streaming and large batch jobs. Checkpointing captures operator state and in-flight progress. In streaming, checkpoints are often coordinated with barriers that align progress across partitions. In batch, recomputation from input is possible, but expensive if input is remote or if intermediate results are large. In unified execution, both replay and live processing may be in flight, and checkpoint decisions affect both [44]. The runtime can represent progress as a combination of input offsets for sources and watermark frontiers for event time. A consistent checkpoint records the

set of source offsets and state snapshots such that replay from offsets will reproduce the same state transitions given deterministic processing and the same watermark policy.

Operator fusion complicates checkpointing because multiple logical operators share a kernel. The runtime can still checkpoint state at operator boundaries by structuring the fused kernel with explicit state modules, each with serialization and restore interfaces. The checkpoint barrier triggers each module to snapshot its state [45]. To avoid pausing the whole kernel, incremental snapshots can be used, where state is copied in the background while updates are logged. This resembles copy-on-write: when a state entry is modified during snapshot, the old version is preserved. The overhead depends on write amplification. Under high update rates, the runtime may adjust checkpoint frequency to reduce overhead, trading longer recovery time for lower steady-state latency. Declarative backpressure can set constraints on maximum recovery point objective or on checkpoint overhead, turning this into a controlled trade-off rather than an implicit engineering choice [46].

Replay mechanics unify batch and stream recovery. If a failure occurs, the runtime restores state from the last checkpoint and replays input from recorded offsets. For batch replay segments, this is equivalent to rerunning a suffix of the batch scan. For live streams, it reprocesses buffered data. A unified runtime can exploit the fact that batch segments are typically stored in distributed storage with high throughput [47]. It can accelerate recovery by temporarily allocating more resources to replay, but only if this does not violate live SLOs after recovery. Declarative backpressure can specify recovery priorities, such as “restore low-latency service first, then catch up backlog,” which compiles into a staged recovery schedule.

Consistency models matter for outputs. Some sinks require exactly-once semantics, others tolerate at-least-once with idempotence. The runtime can include sink semantics as annotations in the IR, enabling the optimizer to choose checkpointing and commit protocols [48]. For example, sinks that support transactional commits can align commits with checkpoints. Under fusion, sink writes may be embedded in the same kernel; the runtime must ensure commit ordering remains correct. This is handled by separating side effects into an effect log that is flushed and committed only when the checkpoint succeeds, keeping the dataflow deterministic.

Distributed co-optimization includes placement. Operators can be placed near data sources or near state stores [49]. For batch scans, placing compute near storage reduces network. For streaming, placing compute near ingress reduces latency. When both occur, the runtime may split placement by origin, or it may co-locate to share caches and state. The placement problem can be formulated as minimizing a combination of network cost and latency under capacity constraints. Let $x_{o,m} \in \{0,1\}$ indicate operator o placed on machine m. The network cost can be expressed as [50]

$$C_{\text{net}} = \sum_{(o \to o') \in E} \sum_{m,m'} x_{o,m}\, x_{o',m'}\, \lambda_{o \to o'}\, b_{o \to o'}\, d_{m,m'}, \tag{5.1}$$

where b is bytes per tuple and $d_{m,m'}$ is a distance cost. Capacity constraints enforce CPU and memory bounds on each machine. This is a combinatorial optimization; the runtime uses heuristics such as greedy placement with local improvements, guided by measured traffic. Declarative backpressure influences placement by emphasizing certain paths and by restricting migration frequency to preserve stability.

6. Co-Optimization Algorithms, Approximate Analytics, and Reproducible Evaluation

The core technical challenge of batch–stream co-optimization is that multiple decision layers interact: logical plan rewriting, physical operator selection, fusion boundaries, placement, rate control, checkpoint scheduling, and approximation parameters. Optimizing them jointly is combinatorial and dynamic. A unified runtime therefore uses a layered approach where an offline or slow-time-scale optimizer chooses structural decisions, while an online controller adjusts continuous knobs in response to load [51]. Even then, the layers must share a consistent objective and constraints expressed by the declarative backpressure contract.

Structural optimization begins with query planning over the unified IR. Traditional join ordering and predicate pushdown can be adapted to temporal relations by including time and state constraints. For example, pushing a filter earlier reduces state size in downstream aggregates, which may be critical for streaming memory. However, pushing a filter across a watermark-dependent operator can change when results become final if the filter depends on late-arriving attributes [52]. The optimizer uses legality rules derived from operator semantics. Within the legal space, a dynamic programming approach can enumerate join trees for subgraphs that are sufficiently small. The DP state includes estimated cardinalities and selectivities as functions of time, which can be represented by parametric models fitted from recent observations. For larger graphs, the runtime uses heuristic search such as iterative improvement guided by a cost model.

Operator selection includes choosing between exact and approximate implementations [53]. For example, distinct counting can be exact with a hash set or approximate with a sketch. In streaming, exact can be infeasible due to state growth. The runtime treats approximation parameters as variables with error bounds. Error analysis is integrated into optimization so that plans are chosen to satisfy an accuracy constraint when possible and to degrade gracefully otherwise. If an approximate operator produces an estimate $\hat{y}$ of a true value y with bound $|\hat{y} - y| \leq \epsilon$ with probability at least $1 - \delta$, then downstream operators that depend on $\hat{y}$ can propagate error bounds. For linear downstream transformations, bounds propagate linearly; for nonlinear transformations, the runtime can use Lipschitz constants or local derivatives to bound error amplification [54]. If a downstream operator computes $f(\hat{y})$, then a first-order bound is

$$|f(\hat{y}) - f(y)| \;\leq\; \sup_{\xi \in [y-\epsilon, y+\epsilon]} |f'(\xi)|\, \epsilon, \tag{6.1}$$

assuming differentiability. This aligns with backprop-friendly modeling: derivatives used for error propagation are also useful for tuning.

Approximate joins and similarity search introduce additional complexity. If a join predicate is based on similarity in an embedding space, approximate nearest-neighbor methods such as hashing-based indexing can be used. A locality-sensitive hashing family yields a collision probability that depends on distance, enabling probabilistic recall guarantees [55]. The runtime can expose a knob controlling the number of hash tables or probes, trading recall for latency. Under pressure, it can reduce probes, which reduces CPU and latency while potentially missing matches. Declarative backpressure can constrain the maximum allowable drop in recall or can allow it when tail latency exceeds an SLO. This turns approximate query behavior into a governed policy rather than an ad hoc emergency measure.

Multi-objective optimization is central because system goals are rarely single-dimensional [56]. Tail latency, throughput, cost, and accuracy form a trade-off surface. A weighted-sum objective is simple but can miss non-convex Pareto regions. A runtime can approximate the Pareto frontier by solving multiple scalarizations and selecting a solution based on policy and current conditions. In practice, the runtime maintains a small set of candidate configurations and uses online bandit-like selection to choose among them. Each candidate corresponds to a fusion plan, a placement, and a set of control knobs [57]. The runtime observes performance and updates confidence. A Bayesian-ish selection strategy can model rewards with uncertainty, preferring configurations that are promising but not yet well explored, subject to safety constraints from backpressure contracts.

Online control adjusts continuous knobs, including admission rates and approximation parameters. A control interval of length Δ collects metrics and updates controls. Let x_t be the vector of controls at interval t and let y_t be the observed signals [**?**]. A linearized dynamics model can be written as

$$y_{t+1} \;\approx\; A_t y_t \;+\; B_t x_t \;+\; \epsilon_t, \tag{6.2}$$

where A_t and B_t are estimated online and ϵ_t captures noise. Model predictive control can then choose x_t to minimize predicted objective over a horizon while satisfying constraints. This is computationally

heavy, but with small control dimension and short horizons it can be feasible. Alternatively, a primal-dual gradient approach updates controls using approximate gradients of constraint violations [58]. To improve stability, the runtime can incorporate integral terms on constraint errors, akin to PI control, and can enforce rate limits on control changes to prevent thrashing.

Co-optimization must also account for energy and cost. If the runtime runs in a cluster with autoscaling, it can trade resource allocation against latency and throughput. An energy model can be approximated as $E = \sum_m (P_{\text{idle},m} + P_{\text{dyn},m} u_m)\Delta$, where u_m is utilization and $P_{\text{dyn},m}$ is dynamic power slope. The runtime can include energy in the objective and impose a budget constraint. This yields a Lagrangian where the multiplier on energy budget influences how aggressively backpressure throttles replay or enables approximation.

Complexity analysis provides insight into algorithmic bottlenecks [59]. For a keyed aggregation with n input records and k distinct keys, hash-based update is expected $O(1)$ per record, but memory is $O(k)$. For a join between streams with windowed state sizes k_1 and k_2, a naive nested loop is $O(k_1 k_2)$ per window, while hash join is $O(k_1 + k_2)$ to build and $O(n)$ to probe, with caveats for skew. In distributed settings, shuffle cost is $O(n)$ in data volume but with constants determined by serialization and network. Operator fusion can reduce constants but may increase code size and instruction cache pressure, which can be modeled by a term proportional to kernel size. Such models inform the optimizer when fusion yields diminishing returns [60].

Query planning with time and state can be framed as a graph optimization problem. Selecting a set of materialization points to minimize cost under memory constraints resembles selecting cut edges in a DAG. If materialization points are represented as binary variables, the optimization resembles a constrained cut problem. Intuition suggests hardness by encoding known difficult partitioning tasks. The runtime uses approximation heuristics [61]. One heuristic is to compute a topological order and greedily fuse along edges with high benefit-to-risk ratio, where benefit is reduced overhead and risk is increased recovery cost or reduced observability. Local search can then adjust boundaries based on measured performance.

Reproducibility and evaluation require deliberate design because adaptive systems can be hard to benchmark. A unified runtime can separate deterministic dataflow semantics from adaptive control decisions by logging control actions and random seeds. Replaying a run with the same control log should reproduce outputs exactly in exact mode and within specified error bounds in approximate mode [62]. For evaluation, workloads must include both batch and streaming components with controllable skew, burstiness, and out-of-order patterns. A workload generator can define a base distribution over keys and times, then apply transformations such as periodic spikes, diurnal cycles, and correlated key hotness. The generator should also produce ground truth for accuracy evaluation, such as exact aggregates for selected windows, enabling error measurement of approximate operators.

Performance evaluation must report not only averages but also tail behavior and stability. Tail latency quantiles, watermark lag, memory high-water marks, and recovery times after injected faults are relevant [63]. Because fusion and backpressure can change scheduling behavior, CPU utilization alone can be misleading; instruction-level metrics such as cache miss rates and branch mispredictions help interpret results. The runtime can embed lightweight profiling in generated kernels, sampling hardware counters at controlled rates. To avoid perturbation, profiling overhead must be bounded and reported.

Distributed evaluation should include network stress and failure modes. For example, the runtime can inject packet loss or bandwidth caps to test how declarative backpressure responds, and can inject worker failures to test checkpointing and replay [64]. The runtime's behavior under mixed batch replay and live streaming is particularly important: evaluation can measure how quickly live latency recovers when replay is active, and how long it takes to catch up backlog after a failure. These measurements tie back to the backpressure contracts, verifying that the control plane honors declared priorities.

Finally, evaluation should include sensitivity analysis. Because the system uses cost models and online estimation, it is important to assess robustness to model error. One method is to perturb estimated service rates or selectivities and observe control stability [65]. Another is to run with intentionally stale models to simulate sudden workload shifts. Reporting these results clarifies whether co-optimization

relies on fragile assumptions. The goal is not to claim universality, but to delineate the regimes where the unified runtime behaves predictably and where it needs additional safeguards.

7. Conclusion

A unified runtime for batch–stream co-optimization can be organized around a temporal-relational intermediate representation that treats batch and stream inputs uniformly, a declarative formulation of backpressure that compiles into enforceable control policies, and an operator fusion framework that respects time, state, and fault-tolerance boundaries while extracting locality and materialization savings. The unified IR makes explicit the interactions between watermark progress, state retention, partitioning, and storage layout, enabling the optimizer to reason about cross-modal workloads rather than treating replay and live processing as separate worlds [66]. Declarative backpressure reframes congestion from an incidental queue phenomenon into a contract over latency, memory, energy, and accuracy, allowing controlled degradation via admission control and approximation knobs with explicit error accounting. Operator fusion, when combined with storage-aware execution and distributed scheduling, can provide significant constant-factor improvements while remaining compatible with checkpointing and recovery through explicit state modules and internal observability hooks.

Co-optimization remains inherently multi-objective and partially combinatorial. A practical approach combines structural heuristics for plan rewrites, fusion, and placement with online control for rates and approximation parameters, using constraint-based formulations and, where useful, differentiable cost surrogates and uncertainty-aware estimation. Evaluation and reproducibility require logging of control actions and careful workload design that exercises skew, burstiness, out-of-order arrivals, and failures. The resulting blueprint clarifies how a single runtime can align execution, control, and optimization across batch and stream, while making policy choices explicit and measurable rather than emergent and opaque [67].

References

[1] G. Farina, “Tractable reliable communication in compromised networks,” 12 2020.

[2] P. Weisenburger, M. Köhler, and G. Salvaneschi, “Distributed system development with scalaloci,” 9 2018.

[3] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst, “Visualizing distributed system executions,” *ACM Transactions on Software Engineering and Methodology*, vol. 29, pp. 1–38, 3 2020.

[4] S. Sakr and A. Y. Zomaya, *Distributed Systems*, pp. 690–690. Springer International Publishing, 2 2019.

[5] “Proceedings of the 1st international symposium on parallel computing and distributed systems,” in *2024 International Symposium on Parallel Computing and Distributed Systems (PCDS)*, pp. 1–1, IEEE, 9 2024.

[6] 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.

[7] D. Przytarski, C. Stach, C. Gritti, and B. Mitschang, “Query processing in blockchain systems: Current state and future challenges,” *Future Internet*, vol. 14, pp. 1–1, 12 2021.

[8] A. Estevan, “An approach to distributed systems from orderings and representability,” *Bulletin of the Iranian Mathematical Society*, vol. 50, 4 2024.

[9] P. C. Sachin and D. Amit, *A Survey on Byzantine Agreement Algorithms in Distributed Systems*, pp. 495–506. Germany: Springer Nature Singapore, 5 2022.

[10] J. Tournier, F. Lesueur, F. L. Mouël, L. Guyon, and H. Ben-Hassine, “Iotmap, a modelling system for heterogeneous iot networks,” 6 2020.

[11] Q. Jia, L. Guo, Y. Fang, and G. Wang, “Efficient privacy-preserving machine learning in hierarchical distributed system,” *IEEE transactions on network science and engineering*, vol. 6, pp. 599–612, 7 2018.

[12] P. Raith, S. Nastic, and S. Dustdar, "Simuscale: Optimizing parameters for autoscaling of serverless edge functions through co-simulation," in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pp. 305–315, IEEE, 7 2024.

[13] D. I. Sukhoplyuev and A. N. Nazarov, "Monitoring fault tolerance in distributed systems," *Computational nanotechnology*, vol. 11, pp. 94–106, 9 2024.

[14] A. Fieschi, P. Hirmer, R. Sturm, M. Eisele, and B. Mitschang, "Anonymization use cases for data transfer in the automotive domain," in *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 98–103, IEEE, 3 2023.

[15] L. Bradatsch, O. Miroshkin, N. Trkulja, and F. Kargl, "Zero trust score-based network-level access control in enterprise networks," in *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1422–1429, IEEE, 11 2023.

[16] J. Pennekamp, R. Matzutt, S. S. Kanhere, J. Hiller, and K. Wehrle, "The road to accountable and dependable manufacturing," *Automation*, vol. 2, pp. 202–219, 9 2021.

[17] H. H. Ali and A. D. S. H. Shaker, "Techniques for secure distributed systems," *Journal of Physics: Conference Series*, vol. 1530, pp. 012006–, 5 2020.

[18] K. Hazelwood, "Accelerator real-time edge ai for distributed systems (reads) disentangling beam losses in the fermilab main injector enclosure using real-time edge ai," in *Accelerator Real-Time Edge AI for Distributed Systems (READS) Disentangling Beam Losses in the Fermilab Main Injector Enclosure Using Real-time Edge AI*, US DOE, 2 2024.

[19] P. Wichmann, A. Groddeck, and H. Federrath, "Fileuploadchecker: Detecting and sanitizing malicious file uploads in web applications at the request level," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pp. 1–10, ACM, 8 2022.

[20] 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.

[21] A. Poshtkohi and M. B. Ghaznavi-Ghoushchi, *Advanced Operating System Concepts in Distributed Systems Design*, pp. 23–43. Auerbach Publications, 2 2023.

[22] "Demystifying distributed systems in cloud-native environments," *Journal of Computational Analysis and Applications*, vol. 34, 10 2025.

[23] T. He and R. Buyya, "A taxonomy of live migration management in cloud computing," *ACM Computing Surveys*, vol. 56, pp. 1–33, 10 2023.

[24] S. Wang, S. Liu, X. Fan, H. Wang, Y. Zhou, H. Gao, H. Lu, and X. Deng, "Vg-net: Sensor time series anomaly detection with joint variational autoencoder and graph neural network," in *2024 IEEE Smart World Congress (SWC)*, pp. 456–463, IEEE, 12 2024.

[25] "Three-layer distributed system based on bayesian classifier," *Distributed Processing System*, vol. 3, 10 2022.

[26] C. Capova, A. Morichetta, A. Lackinger, and S. Dustdar, "Intent-to-learning translation for computing continuum management," in *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*, pp. 1–6, IEEE, 9 2025.

[27] D. Schwachhofer, P. Domanski, S. Becker, S. Wagner, M. Sauer, D. Pflüger, and I. Polian, "Large language model-based optimization for system-level test program generation," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, IEEE, 10 2024.

[28] A. Poshtkohi and M. B. Ghaznavi-Ghoushchi, *IoT and Distributed Systems*, pp. 15–22. Auerbach Publications, 2 2023.

[29] Z. Dong, Z. Wang, X. Zhang, X. Xu, C. Zhao, H. Chen, A. Panda, and J. Li, "Fine-grained re-execution for efficient batched commit of distributed transactions," *Proceedings of the VLDB Endowment*, vol. 16, pp. 1930–1943, 6 2023.

[30] M. Zaccarini, F. Poltronieri, D. Borsatti, W. Cerroni, L. Foschini, G. Y. Grabarnik, D. Scotece, L. Shwartz, C. Stefanelli, and M. Tortonesi, "Chaos engineering based kubernetes pod rescheduling through deep sets and reinforcement learning," in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, pp. 1–7, IEEE, 5 2025.

[31] R. Lichtenthäler, M. Prechtl, C. Schwille, T. Schwartz, P. Cezanne, and G. Wirtz, "Requirements for a model-driven cloud-native migration of monolithic web-based applications," *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, pp. 89–100, 8 2019.

[32] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott, "Programming open distributed systems in maude," in *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming*, pp. 1–12, ACM, 9 2024.

[33] C. M. Zurita and F. Assis, "Hybrid control strategies for greenhouse climate regulation: Pid, fuzzy, and neuro-fuzzy comparative implementation in temperate-dry crop systems," in *2025 XV Symposium on Computing Systems Engineering (SBESC)*, pp. 1–6, IEEE, 11 2025.

[34] 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.

[35] J. Slak and G. Kosec, "Adaptive rbf-fd method for poisson's equation," *WIT transactions on engineering sciences*, vol. 126, pp. 149–157, 9 2019.

[36] "Distributed system model for key management," *Bulletin of TUIT: Management and Communication Technologies*, pp. 1–5, 1 2018.

[37] K. N. Mishra, *Deadlock Prevention in Single-Server Multi-CS Distributed Systems Using Voting- and Priority-Based Strategies*, pp. 115–123. Springer Singapore, 4 2018.

[38] F. D. Gregorio, A. Gamatié, G. Sassatelli, A. Castelltort, and M. Robert, "Exploration of energy-proportional distributed systems," 6 2019.

[39] J. A. McDougall, A. Brighente, A. Kunstmann, N. Zapatka, H. Schambach, and H. Federrath, "Probing with a generic mac address: An alternative to mac address randomisation," in *2024 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pp. 1–6, IEEE, 9 2024.

[40] A. M. Ahmed, M. A. Saeed, A. A. Hamood, A. A. Alazab, and K. A. Ahmed, "Comparative study of static analysis and machine learning approaches for detecting android banking malware," in *2023 3rd International Conference on Emerging Smart Technologies and Applications (eSmarTA)*, pp. 1–8, IEEE, 10 2023.

[41] Z. Wang, H. Chen, Y. Wang, C. Tang, and H. Wang, "The concurrent learned indexes for multicore data storage," *ACM Transactions on Storage*, vol. 18, pp. 1–35, 1 2022.

[42] 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.

[43] N. V. Mokrova, A. M. Mokrov, and A. V. Safonova, "The distributed systems of engineering supervision of permanent structures," *IOP Conference Series: Materials Science and Engineering*, vol. 365, pp. 062036–, 6 2018.

[44] O. Bibartiu, F. Dürr, K. Rothermel, B. Ottenwälder, and A. Grau, "Availability analysis of redundant and replicated cloud services with bayesian networks," *Quality and Reliability Engineering International*, vol. 40, pp. 561–584, 7 2023.

[45] P. Weisenburger, M. Köhler, and G. Salvaneschi, "Distributed system development with scalaloci," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 129–30, 10 2018.

[46] S. Dahdal, F. Poltronieri, A. Gilli, M. Tortonesi, R. Fronteddu, R. Galliera, and N. Suri, "Roamml: Distributed machine learning at the tactical edge," in *MILCOM 2023 - 2023 IEEE Military Communications Conference (MILCOM)*, pp. 33–38, IEEE, 10 2023.

[47] C. Gao, Y. Zhong, and X. He, *Safety Control of Distributed Systems*, pp. 153–159. Elsevier, 1 2024.

[48] S. Ahriz, N. Benmoussa, A. E. Yamami, K. Mansouri, and M. Qbadou, "An elaboration of a strategic alignment model of university information systems based on sam model," *Engineering, Technology & Applied Science Research*, vol. 8, pp. 2471–2476, 2 2018.

[49] *Clock Synchronization in Distributed Systems*, pp. 281–292. CRC Press, 10 2018.

[50] 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.

[51] P. M. Moretti, *Approximations for Distributed Systems*, pp. 55–68. CRC Press, 9 2024.

[52] P. Jeswani, "Transparency and security issues in distributed system," 4 2018.

[53] S. Shankar, "The canonical controller for distributed systems," *Multidimensional Systems and Signal Processing*, vol. 32, pp. 303–311, 8 2020.

[54] V. Podile, N. Kulshrestha, S. Goswami, L. Durga, B. Rachanasree, T. P. Reddy, and P. S. Sarojini, "1the future of business management with the power of distributed systems and computing," 3 2024.

[55] C. Tang, Z. Wang, J. Li, and H. Chen, "Sonata: Multi-database transactions made fast and serializable," *Proceedings of the VLDB Endowment*, vol. 18, pp. 3449–3462, 9 2025.

[56] A. Thakur, S. Verma, N. Sindhwani*, and R. Vashisth, "Applications of optimized distributed systems in healthcare," 3 2024.

[57] M. Bisen, "Integration technology of distributed system based on multi-objective hotopy algorithm," *Distributed Processing System*, vol. 3, 7 2022.

[58] Z. Lu, W. Yu, P. Xu, W. Wang, J. Zhang, and D. Feng, "An ntt/intt accelerator with ultra-high throughput and area efficiency for fhe," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pp. 1–6, ACM, 6 2024.

[59] C. Zhu, S. Wang, X. Fan, X. Deng, S. Liu, Y. He, and C. Wu, "Blockchain-enhanced federated learning for secure and intelligent consumer electronics : An overview," *IEEE Consumer Electronics Magazine*, pp. 1–12, 1 2025.

[60] N. Evangelou-Oost, C. Bannister, and I. J. Hayes, *Contextuality in Distributed Systems*, pp. 52–68. Germany: Springer International Publishing, 3 2023.

[61] S. H. S. A. UBAIDILLAH, N. AHMAD, and N. A. Sahabudin, "A survey on potential reactive fault tolerance approach for distributed systems in big data," in *Third International Conference on Computer Vision and Information Technology (CVIT 2022)*, pp. 21–21, SPIE, 2 2023.

[62] C. Avasalcai, C. Tsigkanos, and S. Dustdar, "Resource management for latency-sensitive iot applications with satisfiability," *IEEE Transactions on Services Computing*, vol. 15, pp. 2982–2993, 9 2022.

[63] A. B. Chernyshev, V. Antonov, and Z. Mayransaev, "Unified sectoral criterion for the stability of distributed systems," in *Proceedings of X All-Russian Scientific Conference "System Synthesis and Applied Synergetics"*, pp. 351–355, Southern Federal University, 10 2021.

[64] 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.

[65] G. Gamal, M. Al-Shaikh, M. A. Saeed, A. G. Hazza'a, A. Alomary, and R. Alshehabi, "Evaluating the performance of machine learning models for dynamic resource allocation in nfv," in *2023 3rd International Conference on Emerging Smart Technologies and Applications (eSmarTA)*, pp. 1–9, IEEE, 10 2023.

[66] L. Su, X. Wang, and L. Wang, "A resilience analysis method for distributed system based on complex network," in *2021 IEEE International Conference on Unmanned Systems (ICUS)*, pp. 238–243, IEEE, 10 2021.

[67] Z. Zhao, M. Wu, H. Chen, and B. Zang, "Characterization and reclamation of frozen garbage in managed faas workloads," in *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 281–297, ACM, 4 2024.