Beyond Storage: State as a Runtime Control Problem in Parallel and Distributed Systems

SHUHAO ZHANG, Huazhong University of Science and Technology, China
HAORAN PENG, Huazhong University of Science and Technology, China
CHUN HO MA, Huazhong University of Science and Technology, China
MAO YANCAN, ByteDance, Singapore

State is increasingly the main source of both performance leverage and system fragility in modern data systems, from stream processors and transactional streaming engines to edge analytics stacks, large language model serving systems, RAG services, and continual-learning pipelines. Yet the relevant literature remains fragmented across contention-aware access control, hardware-conscious execution, LLM memory management, and long-horizon update and retention mechanisms. This survey asks how a parallel or distributed runtime should manage shared, evolving, and performance-critical state under concurrency, heterogeneous hardware, and non-stationary workloads. We organize prior work around three coupled dimensions: state access and scheduling, state-aware execution, and state evolution and reuse. Across these dimensions, we synthesize the literature through a common scaffold based on state object, control surface, coupling path, evaluation boundary, and remaining systems gap. We also use a propagation-oriented perspective to explain how local mismatches amplify into system-level instability. Building on that synthesis, the survey derives cross-domain mechanism comparisons, a contract-oriented blueprint for stateful runtimes, a taxonomy of recurring anti-patterns, and a disturbance-oriented evaluation agenda. The central systems lesson is that state should be treated as a first-class runtime control problem rather than as a passive storage detail.

CCS Concepts: • **Computer systems organization** → **Distributed architectures**; • **Software and its engineering** → *Middleware*; • **Information systems** → *Data management systems*; • **Computing methodologies** → Machine learning.

1 INTRODUCTION

The dominant systems abstractions of the last decade have shifted from stateless throughput engines toward long-running services whose behavior is governed by evolving state. Foundational streaming systems such as MillWheel, Naiad, D-Streams, and the Google Dataflow model already made state and time first-class runtime concerns [7, 8, 157, 241]. A transactional stream processor maintains window contents, indexes, queues, and recovery metadata [151, 256, 257]. An edge analytics pipeline relies on compression dictionaries, approximate intermediate summaries, and quality-aware operator state [245, 246]. An LLM serving runtime maintains per-request KV caches, page tables, prefix-sharing structures, adapter pools, model-residency metadata, and decode scheduling metadata that directly determine throughput and latency [6, 115, 193, 236, 248, 275]. A modern retrieval-augmented or continual-learning service maintains evolving memory, retriever parameters, retained samples, and update traces across rounds of inference [124, 224, 259, 281]. In all of these cases, the decisive systems question is whether the runtime can keep that state governable without losing service stability.

This evolution has made state management in parallel and distributed systems both more important and more difficult. Traditional decomposition often studies operator placement, memory layout, or model adaptation separately. In practice, these concerns interact. A hotspot in shared state access can change queueing behavior and hide locality opportunities. An optimization that improves kernel speed on one processor may fail to improve service-level latency once energy, data

Authors' addresses: Shuhao Zhang, Huazhong University of Science and Technology, Wuhan, China, shuhaozhang@hust.edu.cn; Haoran Peng, Huazhong University of Science and Technology, Wuhan, China; Chun Ho Ma, Huazhong University of Science and Technology, Wuhan, China; Mao Yancan, ByteDance, Singapore, Singapore, maoyancan@u.nus.edu.

movement, or accuracy constraints are accounted for. A memory update rule that accepts more new information may destabilize retention, retrieval quality, or inference consistency downstream. The systems challenge is therefore not merely to optimize state in one phase, but to keep a coupled state lifecycle governable as local gains propagate into later constraints.

The discussion adopts that propagation view. We use *state management* to mean the mechanisms that govern how shared state is organized, observed, accessed, updated, retained, and reused across execution boundaries. Under that definition, the same control pattern reappears across classic streaming systems, hardware-conscious data processing, and newer AI runtimes. In a multicore streaming engine, the key object may be a shared hash table, window buffer, or recovery log. In an edge analytics runtime, it may be a compression dictionary or bounded error-compensation state. In an LLM serving runtime, it may be the KV cache together with the allocator, page table, prefix tree, or reuse index that governs who can touch it and when [115, 275]. In a retrieval-augmented service, it may be a retriever memory, vector index, or retained coreset. The object changes, but the systems obligations do not: make it visible to the runtime, characterize its access cost, couple it to execution policy, and ensure that updates do not destroy future reuse value.

Our synthesis is organized around three dimensions. They are analytically distinct but operationally coupled, which is why the later sections keep returning to how decisions along one dimension reshape the others. The purpose of this decomposition is not to claim that real systems can separate access, execution, and evolution cleanly, but to make visible which control decision is dominant at a given point and how that decision later reappears as a constraint on the other two axes.

(1) **State access and scheduling**: how systems expose contention, model access cost, and schedule work around hotspots, topology, and recovery requirements.
(2) **State-aware execution optimization**: how systems turn state layout and execution coupling into stable end-to-end gains under hardware, energy, latency, and quality constraints.
(3) **State evolution and reuse**: how systems continuously write, retain, organize, and reuse state for dynamic learning and inference.

Figure 1 summarizes the organizing perspective used throughout the survey. Rather than viewing access, execution, and evolution as separate stages, we treat them as one control loop over long-lived state. That framing also suggests the comparison standard used here: in this survey, a paper is treated as advancing state management when it makes a control decision over that loop more observable, analyzable, or governable.

The survey makes four concrete contributions to that problem framing. It introduces a unified vocabulary spanning streaming, approximate execution, serving, retrieval, and memory-centric intelligent services; develops a taxonomy and comparative synthesis organized around access, execution, and evolution; distills recurring anti-patterns into a blueprint-oriented view of runtime contracts and disturbance-aware evaluation; and derives a research agenda for integrated runtimes that coordinate observability, scheduling, update control, recovery, and reusable memory services on heterogeneous hardware. The emphasis is comparative and integrative rather than exhaustive: the manuscript prioritizes recurring control seams and transfer-worthy mechanisms over complete subfield-by-subfield inventories. It also treats evidence asymmetrically on purpose: mechanisms that expose a reusable control abstraction but leave disturbance robustness or cross-controller composition unresolved are framed as partial closures, not as finished runtime solutions.

The rest of the paper is organized as follows. Section 2 establishes the survey scope, clarifies the analytical model, and introduces the taxonomy and comparison scaffold used throughout the manuscript. Section 3 develops the three core dimensions of state management, covering access and scheduling, state-aware execution, and state evolution and reuse. Section 4 then synthesizes

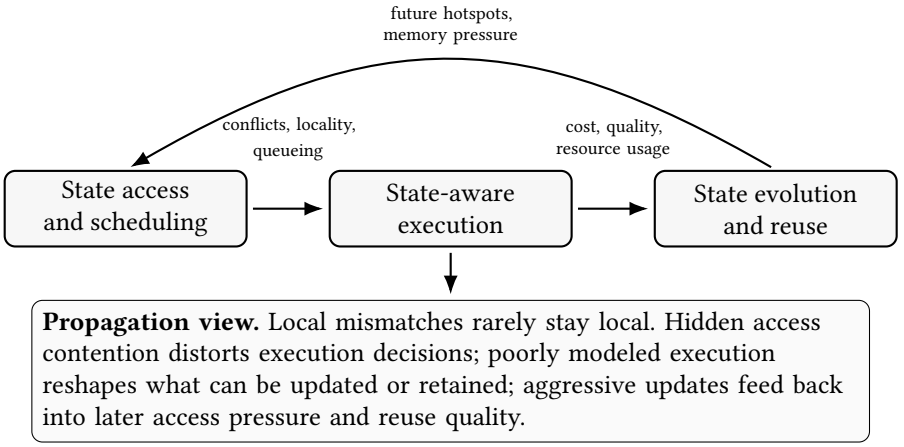

Fig. 1. A propagation-oriented view of state management. The survey's core claim is that access, execution, and evolution are coupled through feedback rather than separable pipeline stages.

the literature comparatively across domains and extracts the conditions under which mechanisms remain transferable beyond one workload family. Section 5 distills the resulting design implications into architectural principles, anti-patterns, and a contract-oriented blueprint for stateful runtimes. Section 6 turns those implications into an evaluation vocabulary and forward-looking research agenda, and Section 7 concludes.

2 FOUNDATIONS: SCOPE, MODEL, AND TAXONOMY

This section defines the survey boundary, clarifies which papers count as core systems evidence, and explains why the manuscript compares mechanisms by control seams rather than by subfield chronology. It also establishes the taxonomy and five-part scaffold that serve as the common comparison contract for all later sections.

We focus on systems in which state is both shared and performance-critical. This includes stream processing engines, transactional streaming systems, edge and heterogeneous analytics runtimes, approximate execution frameworks with quality constraints, and dynamic AI services such as continual learning and RAG. We exclude purely static database storage engines, purely model-centric training methods that do not expose runtime state-management mechanisms, and application papers where state is incidental rather than central.

Our source material combines representative systems papers, journal articles, and recent intelligent-service systems papers. We assemble that corpus by starting from recurring control problems rather than from one system lineage: shared-state access and scheduling, hardware-conscious execution, and evolving memory services provide one anchor set [151, 224, 259, 260, 263], while foundational streaming systems [7, 8, 157, 241], stateful serving runtimes for LLM inference [6, 115, 275], recovery systems, compression and approximation frameworks, and dynamic retrieval services provide the adjacent traditions needed to test whether those mechanisms transfer across domains. This selection strategy keeps the survey comparative rather than genealogical: papers enter the analytical core when they externalize a runtime control seam over state, and they remain contextual when they primarily motivate workloads or models without exposing a governable systems mechanism.

We use three inclusion criteria. First, the system must surface state as an explicit runtime concern rather than a hidden storage detail. Second, the paper must expose a systems mechanism such as scheduling, cost modeling, update control, migration, retention, or reuse orchestration. Third, the evaluation must connect that mechanism to end-to-end properties such as throughput, tail latency, recovery time, quality degradation, or long-horizon reuse quality. These criteria intentionally bias the survey toward papers that make control decisions legible. They also explain why some influential application papers appear only as context: many use state, but fewer make state management the main systems abstraction. For retrieval and learning lines in particular, model-centric papers enter the analytical core only when they externalize managed memory objects, planner-visible controls, or explicit maintenance boundaries. Otherwise, they serve as boundary-setting context rather than primary systems evidence.

2.1 Survey Protocol and Evidence Posture

The review protocol used here is comparative and mechanism-centered rather than bibliometric in the narrow sense. We seed the corpus from canonical systems lines and adjacent surveys, then expand it by following forward and backward citation paths around recurring control problems: shared-state access and scheduling, state-aware execution and placement, and state evolution, refresh, and reuse. That procedure is intentionally designed to complete mechanism families rather than to maximize raw paper counts. It also means that the manuscript does not claim exhaustive coverage of every application line that uses state. Instead, it aims for auditable coverage of the papers that make state-management control surfaces explicit enough to compare across domains.

This protocol also imposes two evidence filters. First, when a stable venue version exists, we prefer it over an arXiv version so that the synthesis rests on a citable archival record rather than on a moving preprint target. Second, recent papers are included into the analytical core only when they expose a new state object, a new runtime control seam, or a new evaluation boundary that changes the comparative argument. Papers that mainly provide workload motivation, model improvements, or deployment anecdotes remain contextual even if they are influential in their own subfield. The same discipline applies to newly emerging 2025–2026 systems: they are useful when they reveal unresolved lifecycle debt, transfer costs, or controller-composition issues, but they should not automatically be read as evidence of a settled runtime architecture.

Accordingly, the manuscript uses an explicit evidence posture throughout the later synthesis. Foundational papers establish a state object and make its immediate control boundary legible; transfer-oriented systems show that a mechanism survives at least one neighboring disturbance regime, hardware path, or service boundary; frontier systems often expose the next unresolved lifecycle seam without yet closing the full runtime loop. These categories are not labels of paper quality. They are a way to avoid flattening canonical results and still-emerging mechanisms into one undifferentiated body of evidence. The comparative sections therefore treat strong local speedups, transfer demonstrations, and long-horizon governance claims as different kinds of evidence rather than as interchangeable proof of maturity.

2.2 Positioning Relative to Existing Surveys

Several prior surveys already cover important slices of this design space. Stream-processing and complex-event-processing catalogs organize operator-level and query-level design choices inside streaming engines [36, 78]. The transactional stream processing survey focuses on execution models, correctness tradeoffs, and system structure within one specialized stateful runtime family [260]. The hardware-conscious stream processing survey concentrates on processor-aware optimization techniques for streaming workloads [263]. The continual learning survey instead organizes anti-

forgetting mechanisms around learning settings, benchmarks, and model behavior rather than runtime state governance [38].

These are valuable adjacent references, but they adopt narrower units of analysis than the present manuscript. This survey differs in both comparison unit and cross-domain ambition. Our organizing question is not limited to one application class, one execution substrate, or one learning setup. Instead, we compare systems by the control problem they expose: what the dominant state object is, which runtime control surface is available over it, how local decisions propagate system-wide, and which service boundary is actually optimized.

That choice allows us to place migration, checkpointing, KV-cache lifecycle control, vector-index maintenance, retrieval freshness governance, and retention budgeting into the same analytical frame without flattening them into a chronology of subfields. The manuscript therefore does not attempt to replace the specialized surveys above. Instead, it asks which systems abstractions recur once state becomes the primary object of runtime control across streaming, serving, retrieval, and long-horizon memory services. This is also why some application-facing or model-facing papers appear only as context here, whereas papers that expose stateful control loops occupy the analytical core.

2.3 Broader Literature Landscape

The broader systems literature matters here for a narrower reason than simple coverage. Efficient state management did not emerge from one subcommunity, but from several lines that repeatedly exposed the same runtime seams under different names: when state becomes visible enough to schedule, when ownership becomes transferable enough to rebalance or recover, when correctness constrains what may be shared or replayed, and when update or movement debt starts to dominate later execution. The purpose of this subsection is therefore not to catalog every adjacent area, but to show why the survey's later three-axis decomposition—access and scheduling, execution optimization, and evolution and reuse—has a stable systems basis.

The first line comes from stream and dataflow systems, where state became visible as an execution object before it became a unifying survey object. Aurora and Borealis made operator state and adaptation explicit inside continuous-query plans; Flink, Trill, Differential Dataflow, StreamCloud, Structured Streaming, and related systems then showed that progress tracking, incremental maintenance, elasticity, and recovery all reshape the same state boundary rather than living in isolated subsystems [1, 2, 11, 19, 20, 23, 36, 69, 78, 153]. The durable lesson from this lineage is not simply that stream processors keep state, but that performance and correctness both depend on when state is exposed, who currently owns it, and how safely it can move under disturbance. That is the historical root of the access-and-scheduling axis used later in the manuscript.

The second line comes from transactional, replicated, and memory-resident data systems, which make a different point that the survey will reuse repeatedly: state is not only a performance artifact but also a correctness boundary. H-Store, Spanner, Calvin, RAMCloud, FaRM, and later high-performance concurrency-control systems treated state as a jointly governed object spanning consistency, placement, logging, and remote access, while classic snapshot, replication, and state-machine-replication results clarified what may be migrated, replayed, or shared without violating service guarantees [24, 33, 42, 102, 164, 165, 183, 190, 204, 237, 242]. This distributed-systems thread is important throughout the survey because later notions such as ownership transfer, shard exposure, and short-lived memory reuse are still constrained by the same question of which state transitions remain legal at the service boundary.

The third line comes from serving and memory-governance systems, where state becomes the hidden scheduler of the whole runtime. Clipper and Clockwork already treated caching, batching, predictability, and dispatch as runtime controls [35, 68]; modern LLM-serving systems made the

relevant state objects much more explicit through KV pages, adapter state, model residency, and prefill/decode separation [6, 30, 34, 79, 115, 126, 172, 186, 192, 193, 195, 234, 236, 248, 275, 278]. A nearby infrastructure literature on tiered and disaggregated memory arrives at the same abstraction from another angle: extra capacity is useful only when the runtime can decide which state must stay near compute and which movement or maintenance debt can be deferred safely [25, 32, 83, 91, 128, 174, 179, 181, 215, 272, 280]. Together these lines motivate the execution-optimization axis as a question of governed placement, representation, and movement rather than kernel speed alone.

The fourth line comes from retrieval, long-horizon memory, continual retention, and approximation-aware systems, which make lifecycle debt visible over longer timescales. REALM, RAG, RETRO, Atlas, Self-RAG, HippoRAG, and RAPTOR show that reusable memory is valuable only when refresh, lookup, and maintenance remain coordinated [12, 14, 74, 90, 120, 189, 218]. Continual-learning and bounded-retention systems expose the same problem under update pressure, where memory budgets, replay choices, and future reuse value must be co-governed [9, 15, 27, 38, 75, 111, 145, 175, 182, 185]. Approximate analytics adds a complementary lesson: once summaries or samples stand in for full state, the runtime must jointly reason about error, update cost, and future reuse [5, 31, 64, 82, 214]. This is the clearest foundation for the evolution-and-reuse axis: state becomes a lifecycle control problem once present choices change future freshness, utility, and service stability.

Several adjacent traditions remain outside the survey core but still help fix the article's analytical boundary. General-purpose dataflow and storage systems such as Dryad, FlumeJava, Bigtable, Dynamo, Cassandra, Megastore, and Percolator show how large services externalize state into programmable, replicated layers with explicit update semantics [13, 22, 26, 39, 89, 116, 158, 173]. CRDT-style replicated objects, dynamic graph systems, vector indexes, and low-level attention kernels each expose one more reason the survey cannot reduce state management to caching alone: merge legality, mutable structure, index maintenance, and memory-traffic shape all become runtime-visible once the state object persists long enough to constrain later control [37, 57, 93, 105, 107, 149, 191]. The role of this broader landscape is therefore bounded but important. It establishes that the rest of the manuscript is not juxtaposing unrelated application trends, but comparing recurring runtime control problems that have long existed across distributed systems, dataflow, storage, serving, and adaptive memory services.

2.4 System Model: Where State Lives in a Runtime

Before defining state as a runtime-control object, we first clarify the system model assumed by this survey. Prior works [20, 21, 150] on stream processing provides a useful methodological template: a system model should first describe the data, computation, deployment, and execution graph, and only then define which internal objects must be exposed as manageable state. For example, SEEP [51] distinguishes the data model, operator model, query model, and query execution model before introducing operator state management primitives for checkpointing, backup, restoration, and partitioning. Although SEEP targets stream processing, this modeling style is more general: state becomes meaningful only after the runtime boundary around computation, communication, scheduling, and recovery has been made explicit.

We therefore view a parallel or distributed system as a runtime that executes a graph of computational units over data items, requests, or events. The computational units may be streaming operators, database transactions, serverless functions, tasks or actors, LLM serving workers, retrieval operators, or agentic reasoning stages. These units are connected by communication paths such as shuffles, RPCs, remote memory transfers, object-store reads, collective communication, KV-cache transfers, or vector-index lookups. A scheduler or control plane assigns work to resources,

decides placement and batching, reacts to load changes, and may trigger migration, checkpointing, eviction, refresh, or recovery. The runtime also maintains a storage or memory substrate that may include local memory, GPU HBM, remote memory, SSDs, distributed logs, checkpoint stores, object stores, or external indexes. In this model, state is not a separate storage module. It is any runtime object that is read, written, moved, retained, or reused across these execution boundaries.

This model lets us compare systems that otherwise appear unrelated. In a stream processor, the computational unit is an operator, communication occurs through streams or shuffles, and state appears as keyed state, windows, indexes, progress metadata, and checkpointed operator state. In a transactional dataflow or database system, the computational unit is a transaction, task, or execution fragment, and state appears as records, indexes, versions, locks, logs, dependency metadata, and recovery snapshots. In a general-purpose distributed runtime such as a task or actor system, state appears as object-store entries, actor-local memory, lineage metadata, placement state, scheduler queues, and checkpointed execution context. In an LLM serving system, the computational units are prefill and decode workers, communication includes model-parallel transfer and KV-cache movement, and state appears as KV caches, page tables, prefix-sharing structures, adapter pools, model-residency metadata, and scheduling metadata. In a RAG or agentic workflow, the computational units are retrievers, planners, tools, and generators, while state appears as vector indexes, document memories, workflow context, tool outputs, retained traces, and memory-update queues.

Across these examples, the concrete object changes, but the runtime relationship is stable. State is coupled to computation because workers must read or update it to make progress. It is coupled to communication because moving work often requires moving, copying, or reconstructing state. It is coupled to scheduling because placement, batching, admission, and migration decisions depend on state size, ownership, hotness, freshness, or reuse potential. It is coupled to memory and storage because the physical representation of state determines locality, capacity pressure, and transfer cost. It is coupled to fault tolerance because the runtime must know which state has become externally visible, which updates can be replayed, and which objects must be restored after failure. Finally, it is coupled to long-horizon service quality because update and retention decisions can change future access patterns, retrieval quality, model behavior, and recovery cost.

This system model also clarifies why state has multiple forms. *Logical state* captures the semantic object maintained by the application, such as a window, table version, KV cache, vector memory, or retained sample set. *Physical state* captures where and how that object is represented, such as hash-table buckets, memory pages, GPU blocks, log segments, checkpoint files, or index nodes. *Metadata state* records ownership, routing, progress, page mappings, lineage, timestamps, or migration status. *Control state* summarizes runtime observations such as hotness, queue length, memory pressure, freshness, cache-hit ratio, or recovery progress. *Evolution state* records how state changes over time through updates, compaction, retention, refresh, eviction, or reuse. These forms are not independent. A change in logical ownership may require physical movement, metadata updates, scheduler decisions, and new recovery constraints. Similarly, a freshness policy for a retriever memory may change vector-index maintenance cost, query latency, and downstream answer quality.

The purpose of this model is not to force all systems into the same implementation template. Rather, it provides a common runtime vocabulary for asking when an object should be treated as managed state. An object becomes relevant to this survey when it satisfies three conditions. First, it persists beyond one isolated function invocation, request step, or operator firing. Second, its value or placement affects future execution, scheduling, recovery, quality, or reuse. Third, the runtime can expose a control surface over it, such as placement, partitioning, migration, checkpointing, admission, batching, eviction, retention, refresh, or reuse orchestration. This definition intentionally

Table 1. State in representative runtime architectures.

System class	Computational unit	Communication path	Representative state objects
Stream processing	Operators and tasks	Streams, shuffles, backpressure channels	Keyed state, windows, operator state, watermarks, checkpoint metadata
Transactional dataflow and databases	Transactions, execution fragments, workers	Dependency exchange, log replication, remote reads	Records, indexes, versions, locks, dependency graphs, logs, recovery snapshots
Distributed task/actor runtimes	Tasks, actors, executors	RPCs, object-store transfers, lineage replay	Objects, actor-local state, lineage metadata, placement state, scheduler queues
LLM serving	Prefill and decode workers	KV transfer, model-parallel communication, request queues	KV caches, page tables, prefix trees, adapter state, model-residency metadata
RAG and agentic workflows	Retrievers, planners, tools, generators	Vector-index queries, tool calls, memory updates	Vector indexes, document memories, workflow context, tool traces, retained memories
Edge and approximate analytics	CPU/GPU operators, sensor tasks	Device transfer, edge-cloud transfer	Summaries, sketches, compression dictionaries, quality-control metadata

covers both long-lived persistent state and short-lived semi-persistent state. A database index and a checkpoint log are state, but so are a KV-cache page, an actor-local object, a prefix-sharing tree, or a retriever-update queue if they shape future runtime decisions.

This architectural view motivates the taxonomy that follows. Since state crosses computation, communication, scheduling, storage, and recovery boundaries, state management cannot be reduced to one subsystem such as storage, caching, or memory allocation. The relevant question is instead which control problem dominates at a given point in the lifecycle. The access-and-scheduling dimension studies how state is exposed, costed, and coordinated under concurrency. The execution-optimization dimension studies how state representation, placement, and movement interact with hardware and service objectives. The evolution-and-reuse dimension studies how state is updated, retained, refreshed, and reused over time. The following sections use this model to compare stream processors, transactional engines, heterogeneous analytics systems, LLM serving runtimes, RAG services, and continual-learning pipelines under a common state-management lens.

2.5 A Unifying View of Stateful Computing Systems

2.5.1 What Counts as State? In this survey, *state* includes any persistent or semi-persistent runtime object whose value influences future execution beyond a single operator invocation. Examples include hash tables and windows in streaming operators, shared indexes and queues in multi-core runtimes, compression dictionaries, adaptive buffering metadata, KV caches and their page-allocation metadata in LLM serving systems, coresets for continual learning, retriever memory,

Table 2. Taxonomy of state-management problems.

Dimension	Core question	Representative concerns
Access and scheduling	How is shared state exposed, costed, and scheduled under concurrency?	hotspot diagnosis, conflict propagation, locality, topology, recovery, migration, prefix-aware sharing
Execution optimization	How does state interact with hardware and service objectives?	placement, compression, KV-cache paging, prefill/decode coupling, approximation, latency-energy-quality trade-offs
Evolution and reuse	How is state updated, retained, and reused over time?	online updates, memory budgets, cache growth and reclamation, sample selection, retriever drift, knowledge maintenance

and structured knowledge stores used by intelligent services. This definition intentionally spans in-memory, persistent, and hybrid forms of state.

Two properties make state systems-hard. First, state is usually shared across tasks, requests, or time windows, so access conflicts propagate. Second, state is temporally extended: the way a system updates state today changes tomorrow's optimization space. These properties mean that local decisions often have delayed and cross-layer consequences. They also explain why the definition above must include semi-persistent and hybrid objects: many of the most important runtime failures emerge not from permanently stored data, but from state whose lifetime is short relative to storage systems yet long enough to reshape later scheduling, quality, or recovery decisions.

2.5.2 The Propagation Perspective. We organize state management in parallel and distributed systems using a propagation perspective. This perspective models a multi-stage control pipeline. A request first encounters *state access*: it must find, read, or synchronize on shared state. The resulting data path determines *state-aware execution*: work is mapped to cores, sockets, accelerators, or approximate operators subject to latency and quality goals. The outputs then contribute to *state evolution*: some state is updated, some is retained, and some is exposed for future reuse. Failures or mismatches in any stage propagate. Uncontrolled contention in access may destroy locality and distort execution modeling. Mis-modeled execution trade-offs may make update policies unsustainable. Over-aggressive updates may increase access pressure and destabilize future inference.

This view differs from traditional layered designs that isolate storage, scheduling, and adaptation. The literature repeatedly shows that state problems cross those layers. Shared access depends on topology-aware execution [257, 262]. Quality-aware execution depends on error-aware state evolution [246]. Continual memory reuse depends on stable update and retention mechanisms [121, 259, 281]. The propagation perspective therefore provides a more faithful abstraction for long-running stateful systems.

2.6 Taxonomy of State Management Problems

Table 2 summarizes the taxonomy used throughout the survey. It should be read as a decomposition of runtime control problems rather than as a catalog of application communities. That distinction matters because the same application may occupy multiple cells over time: a serving stack, for example, can simultaneously expose access skew, execution-level memory pressure, and evolution-level reclamation debt, so classifying by workload label alone would obscure the actual control seams.

Table 3. Reusable analysis scaffold for extending the survey.

Question	What to extract from each paper	Typical answers in this survey
State object	Which runtime object persists across tasks, requests, or time?	windows, indexes, logs, dictionaries, KV caches, retriever memories, coresets
Control surface	What decision does the system make over that object?	placement, batching, migration, paging, admission, retention, reuse orchestration
Coupling path	Why does this state decision propagate to end-to-end behavior?	contention, locality, memory pressure, phase asymmetry, drift, quality-loss amplification
Evaluation boundary	Which system-level property is optimized or bounded?	throughput, p99 latency, recovery time, energy, bounded error, forgetting, reuse quality
Remaining gap	What is still missing after the paper's contribution?	weak observability, no closed loop, limited portability, incomplete lifecycle control, no long-horizon evaluation

The taxonomy is intentionally operational. It does not classify systems by application domain but by the control problem they solve. A stream engine and a RAG service may both belong to the evolution-and-reuse category if the dominant challenge is continuous state updates and future reuse. Likewise, a compression engine and a transactional stream processor may both belong to execution optimization if the central problem is coordinating stateful execution with hardware constraints.

This taxonomy is useful precisely because it is not mutually exclusive at the whole-system level. Mature systems typically occupy more than one row of Table 2. A transactional stream engine may begin as an access-and-scheduling problem and grow into an execution-control problem once topology and recovery costs are modeled explicitly. A retrieval service may begin as an evolution-and-reuse problem but quickly become an access problem when vector indexes, retriever memories, and hot prompts create concentrated contention. The taxonomy should therefore be read as a decomposition of control surfaces, not as a rigid partition of system classes.

2.6.1 A Reusable Analysis Scaffold. The survey applies the same five-question scaffold whenever it summarizes a system line. First, what is the *state object*? Second, what *control surface* does the runtime expose over that state, such as placement, scheduling, migration, admission, retention, or reuse? Third, what *coupling path* makes that control surface important, for example contention, topology, phase asymmetry, drift, or quality decay? Fourth, what *evaluation boundary* does the paper actually optimize, such as throughput, tail latency, recovery time, bounded error, or long-horizon reuse quality? Fifth, what *remaining systems gap* is left open after the paper's mechanism is accounted for?

This scaffold matters for two reasons. It gives the survey a stable comparison unit across otherwise different domains, and it helps prevent a common failure mode in survey writing: turning papers into disconnected summaries of implementation details. Under this template, a stream processor, an LLM-serving runtime, and a retrieval-memory system can all be compared in terms of the state they expose, the runtime decision they support, the service boundary they optimize, and the control gap they leave unresolved.

2.6.2 Representative System Classes. The state-management lens cuts across at least five recurring system classes. The first is *high-throughput streaming and transactional dataflow*, where state ap-

pears as windows, per-key aggregates, dependency graphs, and recovery metadata. The second is *hardware-conscious analytics*, where state appears as compressed representations, scheduling metadata, and approximation structures whose access pattern determines whether hardware speedups survive end-to-end execution. The third is *LLM serving and memory-bound inference*, where state appears as KV caches, page tables, prefix-sharing structures, and structured decoding metadata that must be scheduled under tight memory and latency budgets [6, 115, 275]. The fourth is *continual learning and adaptive services*, where retained samples, coresets, and update traces define both learning cost and future accuracy. The fifth is *memory-centric retrieval and agentic inference*, where retrievers, knowledge structures, and evolving indexes mediate multi-round reuse.

These classes matter because they expose different failure modes. Streaming systems fail by contention amplification, migration cost, or slow recovery. Hardware-conscious analytics fail by misplacing state relative to device topology or by violating quality boundaries when using approximate methods. LLM serving systems fail when KV-cache growth, fragmentation, or prefix-sharing policy turns memory into the primary throughput bottleneck. Continual-learning systems fail by exhausting budgets or forgetting useful history. Memory-centric inference systems fail by update drift, unstable retrieval, or uncontrolled growth in reusable state. Comparing these failure modes matters because the same runtime ideas often reappear under different names.

Figure 2 condenses the manuscript into the single picture that would most naturally sit at the center of a poster. The figure starts from representative system classes, routes them through the survey's three core dimensions, and ends with the blueprint, evaluation, and research agenda that the later sections distill from that comparison.

3 CORE DIMENSIONS OF STATE MANAGEMENT

Up to this point, the manuscript has argued that state should be treated as a runtime control problem rather than as a passive storage detail. The next question is more concrete: once that perspective is adopted, which control problems actually recur across modern systems? This section answers that question through three coupled dimensions—state access and scheduling, state-aware execution, and state evolution and reuse—and uses them to show why a local gain along one dimension often reappears as a downstream constraint on the other two.

To keep the discussion concrete, we use three recurring running examples in this section: (A) a streaming fraud detector under hotspot skew, (B) a multi-tenant LLM assistant under KV-memory pressure, and (C) a RAG knowledge service under corpus drift. Figure 3 summarizes their shared logic: state object, stress event, runtime actuation, and observed service effect.

3.1 State Access and Scheduling

3.1.1 From Invisible Contention to Observable State Access. In running example A, a fraud-detection pipeline suddenly receives a skewed burst after a campaign launch: the model logic is unchanged, but a small set of hot keys now serializes queue and lock paths. The practical question is not "can the operator run faster," but "can the runtime observe and reshape shared access before the skew hardens into replay and migration debt."

As multicore stream and event systems began to scale, a recurring problem became hard to ignore: operator logic itself was often not the main bottleneck, because hidden contention in shared access paths could dominate end-to-end behavior. Studies of multicore stream processing and complex event processing made this visible by showing that shared queues, synchronization patterns, fine-grained operator interactions, and sub-computation sharing can all limit scalability when the runtime cannot see where interference is forming [256, 261]. This line of work established a first principle: state management begins with observability. Systems need path-level metrics that connect hotspots, conflict density, queue growth, and throughput variation; without such

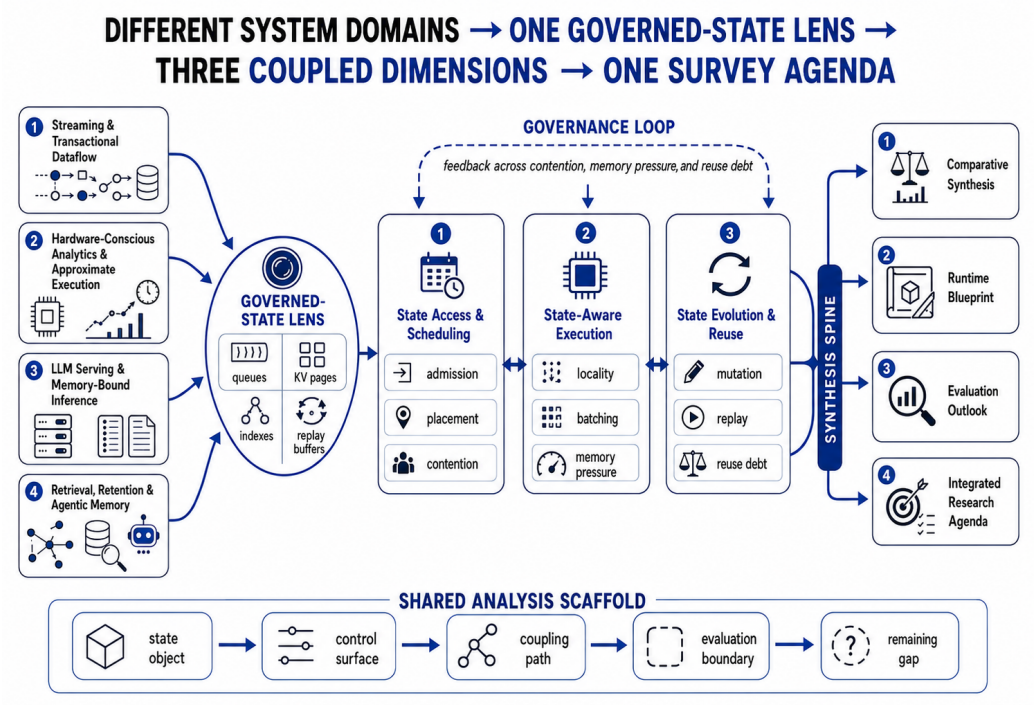

Fig. 2. Overview map of the survey. Representative system domains are unified through one governed-state lens and analyzed along three coupled dimensions: state access and scheduling, state-aware execution, and state evolution and reuse. Their synthesis feeds into four downstream outputs: comparative mechanism synthesis, a runtime blueprint, an evaluation outlook, and an integrated research agenda. The shared analysis scaffold at the bottom shows the common comparison template used throughout the manuscript (state object, control surface, coupling path, evaluation boundary, and remaining gap).

observability, tuning remains heuristic. The enduring lesson is that access metrics matter only when they preserve enough structural context for the runtime to decide whether the right response is repartitioning, migration, admission throttling, or a change in execution grain.

Aurora and Borealis exposed an earlier version of the same problem by making operator graphs and adaptation policies explicit runtime concerns rather than fixed query-plan details [1, 2]. MillWheel, Naiad, and the Dataflow model then clarified that long-running correctness depends on runtime-visible notions of time, pending work, and completion rather than on raw throughput alone [7, 8, 157]. Building on that foundation, later systems made progress and state advancement explicit runtime objects. Trill and Differential Dataflow showed that incremental maintenance is not merely an algebraic optimization: timestamps, differences, and progress frontiers define which parts of state can be advanced, merged, or queried without stalling the whole dataflow [23, 153]. Flink and StreamCloud exposed a related systems lesson from deployment-oriented stream engines: elasticity, operator state placement, and fault handling depend on whether the runtime has a concrete notion of state ownership and movement rather than treating state as opaque operator-local memory [20, 69]. In sequence, these systems define a ladder of control surfaces: operator adaptation first makes state placement explicit, progress semantics determine when state may advance, and ownership transfer determines where it may move. Access observability is

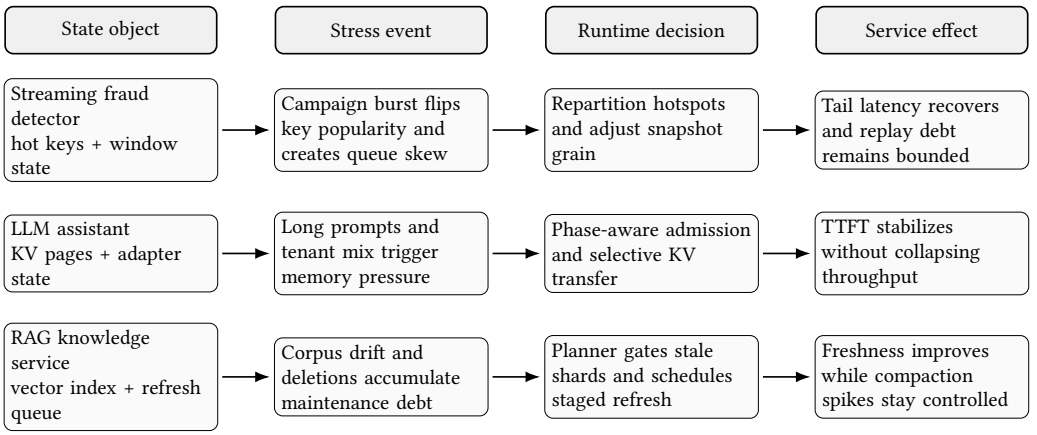

Fig. 3. Running examples used to ground the three core dimensions. Each row shows one concrete service path from state object to disturbance, runtime decision, and service-level outcome. The same control vocabulary is then reused in the literature synthesis.

therefore inseparable from execution progress semantics and transfer semantics: the runtime must know not only where state is hot, but also when it is safe to advance, migrate, or snapshot that state.

3.1.2 Cost Modeling Under Locality and Topology. Once access patterns become observable, the next challenge is pricing them correctly under skew and topology. Topology-aware and concurrent stateful streaming systems showed that hotspot placement, NUMA distance, synchronization design, state partitioning, and task assignment must be considered jointly; locality decisions that ignore any one of these factors can erase the gains of parallelization [257, 262].

These results connect state management to a broader systems theme: locality is not a static placement problem, but a property created jointly by state layout, scheduling, and workload structure. Similar lessons appear in other stateful systems, including unified migration for distributed streaming [150], online interval join systems where shared indexes and ordering constraints interact with runtime concurrency [252], and deployment-time serving planners such as MaveriQ that treat fragmented GPU memory, layer placement, and compression choices as one cost surface rather than as separate provisioning decisions [129]. The experimental study of parallel intra-window joins sharpens the same point from a design-space perspective: execution style, join method, and partitioning scheme each reshape how window state interacts with skew, latency targets, and multicore topology, so there is no workload-agnostic "best" join plan [258]. Across these works, the dominant abstraction is no longer simply a data structure. It is an access-cost surface shaped by contention and hardware placement.

That surface also becomes the engine's future recovery surface. The same partitioning and synchronization choices that improve steady-state locality determine checkpoint grain, replay skew, and migration debt once the workload is disturbed. A scheduler that concentrates hot state onto a small set of cores or partitions may improve average throughput, yet it also creates uneven snapshot cost and slower post-failure convergence. This is why topology-aware placement, concurrency control, and state-transfer design should be compared as one boundary-setting problem rather than as separate optimizations [19, 150, 257, 262, 270].

3.1.3 Runtime Control, Recovery, and Stateful Governance. The mature form of access management is runtime control. MorphStream and its later journal extension are representative: they treat transactional stream processing as a scheduling problem over stateful dependencies, then extend that control surface to non-deterministic state access and multiversioned window management so the engine can adapt execution plans under more irregular workloads rather than rely on fixed static policies [151, 271]. Lightweight asynchronous snapshots in Flink exposed a complementary control surface: checkpoint barriers, in-flight records, and snapshot coordination are part of the same runtime state path, so fault tolerance overhead depends on how consistently the engine aligns these transfer boundaries with normal execution [19]. Follow-up work on fast parallel recovery shows that recovery must be integrated with steady-state execution rather than treated as a separate subsystem [270]. Recovery metadata, log structure, and access control together determine whether a stateful engine can remain efficient after failures.

Earlier elasticity work makes the same point from another angle. SEEP framed elastic event processing as a problem of carrying operator state safely across a changing execution graph, while later operator-state-management work argued that scale-out and fault tolerance should share the same state-transfer machinery rather than be handled by unrelated control paths [21, 51]. Megaphone sharpened the design space by showing that migration grain is itself a control surface: fine-grained bins and progress-aware routing reduce reconfiguration latency, but they also make ownership tracking and transfer progress part of steady-state runtime metadata [160]. More recent unified migration work such as Spacker pushes this line further by treating redistribution as a reusable state-movement substrate instead of a special-case rebalancing procedure, which makes migration policy easier to compare with recovery and placement control [150]. Their shared contribution is to show that operator ownership, transfer granularity, and recovery semantics are inseparable parts of one state-governance problem.

Across these systems, a useful unifying abstraction is an ownership-transfer contract with explicit phases: (i) fence or freeze updates, (ii) transfer a consistent shard snapshot plus progress metadata, (iii) hand off authority with idempotent replay boundaries, and (iv) reopen updates under a new owner. Megaphone's fine-grained bins, Spacker's unified redistribution substrate, and Flink's asynchronous barriers each expose part of this contract, while fast-recovery designs show that replay cost depends on how well those parts align with steady-state scheduling [19, 150, 160, 270]. The remaining gap is that most systems still encode the contract implicitly in separate migration and recovery paths rather than as one explicit handoff protocol.

Adjacent transactional and replicated systems help make the missing fields more explicit. Spanner and Chimera treat authority handoff as lease movement over directories or shared-storage pages, so the critical question is who currently owns the right to expose updates and how long the old owner may remain visible during transfer [33, 84]. Calvin and Raft instead make ordering and epoch change the primary fence: if the next owner inherits a deterministic log position or a higher term, replay and failover stop being ad hoc recovery actions and become legal continuation under a new authority [164, 204]. RAMCloud sharpens the recovery side of the same contract by showing that leader replacement is not enough if the new owner cannot reconstruct serving state quickly from logs and backup replicas [165]. Read together with Megaphone and Spacker, these papers suggest that a reusable handoff protocol needs at least three explicit fields beyond raw state bytes: current authority, replay boundary, and reclamation condition for the old owner.

Recent workflow and training runtimes show that the same governance pattern now extends well beyond classic stream operators. SONIC, ORION, RTSFaaS, Caerus, and SFS all make workflow buffers, leases, or warm-start state scheduler-visible so that placement can optimize ownership continuity across stage boundaries rather than treat each activation as isolated work [54, 147, 148, 251, 269]. Checkpointing and recovery systems such as ByteCheckpoint, FlowCheck, uni-

versal checkpointing, DeCK, MoC, PhoenixOS, CheckFreq, AutoCheck, and Check-n-Run push the same logic into disturbance handling: snapshot timing and persistence grain become controls over recomputation debt, not passive fault-tolerance plumbing [16, 44, 52, 60, 87, 130, 156, 208, 220]. Large-model pipeline and training systems add a complementary coupling path. WeiPipe, Mario, PipeFill, Optimus, MegaScale, TiGon, MEPipe, FBMM, What-if Analysis for Stragglers, ShadowViews, Snowflake, Race, Pegasus, FSMoE, Chimera, NeoMem, WLB-LLM, and SlimPipe all treat bubble state, activation residency, shard skew, or ownership-transfer cost as schedulable runtime objects rather than incidental side effects of parallel execution [10, 50, 84, 85, 96, 99, 122, 127, 133, 134, 141, 168, 200, 201, 207, 217, 282, 284]. Viewed through one mechanism lens, these systems strengthen the survey's main claim: recovery, rebalancing, and utilization control are increasingly one disturbance-aware scheduling problem over shared state, not three loosely related subsystems.

Recent serving-oriented schedulers extend that same argument into latency-facing inference control, but they do so across two distinct boundaries. SOLA, ExeGPT, and ALiSe focus on online serving itself: they expose per-request SLO slack, length skew, and swappable KV residency as scheduling signals so that token-level control can reduce head-of-line blocking and memory-induced latency drift under heterogeneous arrivals [81, 163, 274]. Resource-Multiplexing and Sirius instead study colocated regimes in which serving must share memory and reclaim bandwidth with PEFT or training work, so the key control problem is not only which request runs next, but when ownership of GPU-resident state can be handed off without violating latency targets [77, 210]. Viewed jointly, these systems show that predictive scheduling and cross-workload handoff are two faces of the same governance problem: the runtime must reason about future memory pressure rather than react only after interference materializes. The remaining gap is compositionality. Most systems still optimize one node or one colocated regime at a time, leaving open how these predictive and cross-workload policies should compose with disaggregation, elastic pools, and broader cluster-level ownership contracts.

That lesson remains important today: if migration, checkpointing, and rescaling each maintain their own notion of state ownership, the runtime pays coordination cost repeatedly and recovery semantics become harder to reason about. A stronger design is to treat operator state as a first-class runtime object whose placement, replication, and transfer rules are reused across elasticity, recovery, and fault handling. More broadly, access governance must eventually become a closed loop in which observation feeds cost modeling, cost modeling informs scheduling and recovery, and those choices reshape future observation.

More robust systems in this category therefore stop treating state access as an implementation detail. Once state becomes a scheduling object, new design options open up: conflict-aware batching, topology-sensitive placement, dependency-ordered execution, and recovery plans that are derived from steady-state access structure rather than bolted on afterward. This observation-model-control loop is one of the most reusable patterns in stateful systems research, and the change in abstraction is at least as important as any single algorithmic optimization.

3.1.4 Open Challenges in Access Management. Despite the progress, at least four challenges remain. First, most systems still rely on coarse observability and do not expose state conflicts as first-class runtime objects. Second, topology-aware models often assume relatively stable hardware and workload structure. Third, recovery and migration are still commonly optimized separately from regular access control. Fourth, intelligent-service runtimes increasingly create new shared-state objects, such as vector indexes, retriever caches, and prefix-shared KV caches, whose conflict patterns are poorly understood. Future work should therefore extend access modeling to state objects that are semantic, dynamic, and cross-request.

3.2 State-Aware Execution Optimization

3.2.1 Why Faster Kernels Do Not Guarantee Better Services. Running example B highlights the same point in serving form: a multi-tenant assistant with mixed prompt lengths may have excellent single-kernel speed, yet still miss SLOs once KV residency, prefill/decode phase asymmetry, and adapter multiplexing collide. The runtime wins only when it governs where short-lived state lives and when that state moves.

A common limitation in systems design is to treat stateful execution as equivalent to faster kernels or better operator throughput. The literature reviewed here suggests a more nuanced picture. Once state interacts with heterogeneous hardware, moving data and coordinating access frequently dominate raw compute speed. Moreover, service objectives introduce latency, energy, and quality boundaries that can overturn microbenchmark gains.

Integrated CPU-GPU stream systems make this point concrete. Fine-grained window processing on integrated architectures showed that topology and data path design directly alter the feasible performance envelope, while co-running studies on the same class of hardware reached a similar conclusion from the broader angle of shared-resource interference [202, 249, 250]. These systems emphasize that execution gains emerge only when state organization, data movement, and device mapping are optimized together.

3.2.2 Energy, Compression, and Stateful Data Paths. The CStream line deepens this theme by examining stateful compression on edge and asymmetric multicore devices [244, 245]. Compression dictionaries are state; their access pattern, update granularity, and task decomposition shape both energy and latency outcomes. More importantly, stateful compression creates a multi-objective problem spanning throughput, delay, energy, and compression ratio. That observation generalizes beyond compression: whenever the system carries intermediate state across operators or requests, execution policy becomes a question of which state transitions are affordable on the current hardware path. Device selection, data movement, and representation choice are therefore best viewed as one coupled decision rather than three independent knobs. The same principle is likely to matter even more for modern accelerator-rich inference deployments, where the cost of moving reusable state can dominate the cost of re-executing a kernel.

3.2.3 KV-Cache Management as a State-Execution Problem. Large language model serving makes this point unusually concrete. In these systems, the KV cache is not a minor implementation artifact. It is the dominant short-lived runtime state that grows with prompt length and output length, mediates prefix reuse, and constrains feasible batch size. Once the runtime hits memory pressure, almost every higher-level serving policy becomes a disguised decision about KV ownership, residency, transfer cost, or reclamation timing.

Clipper established an early orchestration baseline in which batching, caching, and model choice are runtime-managed controls rather than fixed application logic [35]. Clockwork sharpened that boundary by treating execution-time predictability as a scheduling signal, turning dispatch and isolation into explicit controls over service-level state rather than best-effort heuristics [68]. Nexus exposed an earlier cluster-level version of the same idea by treating GPU occupancy, batching opportunities, and stage imbalance as runtime-managed serving state rather than static deployment detail [192]. Orca then exposed an early generative variant of this state-centric view by replacing request-level batching with iteration-level scheduling and selective batching, effectively turning partially materialized decode state into a schedulable runtime object instead of hidden per-request baggage [234]. InferLine and INFaaS add adjacent control-plane lessons: end-to-end serving quality already depends on runtime management of queue state, provisioning state, replica placement,

and heterogeneous model choice, so memory governance should be read alongside service-level control loops rather than after them [34, 186].

FlexGen adds a memory-pressure perspective that becomes especially relevant for LLMs: once model and KV state are partially offloaded, transfer schedule and placement policy become part of the serving lifecycle rather than a one-time deployment decision [195]. vLLM then made memory layout explicit by treating KV-cache memory as a paged state object whose allocation and sharing policy determines whether throughput is limited by fragmentation, redundant duplication, and memory waste rather than raw compute [115]. Sarathi-Serve showed that the serving scheduler must also reason about the temporal evolution of that state, because prefill and decode phases create asymmetric pressure on the active cache footprint and therefore on latency-throughput trade-offs [6]. DistServe and Splitwise extend the same logic beyond a single worker: once prefill and decode are disaggregated, the control problem shifts from local cache placement alone to queue isolation, inter-stage state transfer, and goodput-aware admission across heterogeneous pools [172, 278].

Recent KV-compression and token-selection systems refine this memory-pressure argument by changing the representation of the cache itself. QServe and Oaken reduce serving pressure through low-bit or mixed-precision KV paths, but their control surface is not merely quantization: dequantization layout, DMA integration, calibration scope, and batch-size feasibility determine whether memory savings translate into stable serving throughput [109, 135]. Keyformer and Q-Hitter expose a finer-grained retention surface in which token-level cache entries are ranked by recency, attention importance, or quantization sensitivity; the runtime is effectively deciding which historical semantics remain visible to future decoding under a bounded KV budget [3, 268]. VQ-LLM shifts the same tradeoff into vector-quantized codebooks and generated decode kernels, showing that compression is useful only when reconstruction and irregular lookup costs do not erase the capacity gain on the decode path [144]. These works therefore should be read as state-representation controllers, not only as model-compression techniques: they trade cache fidelity, memory footprint, kernel regularity, and future reuse value inside one execution boundary.

Offload and heterogeneous-memory systems expose a complementary control surface. FlexGen treats model and KV placement across memory tiers as an explicit schedule; Neo and AQA push further by making CPU DRAM or remote HBM borrowing part of the online serving path, where swap timing, producer-consumer role selection, and tensor-level residency decisions reshape TTFT, throughput, and long-prompt behavior [97, 112, 195]. Long-context systems such as LServe, LoongServe, and LeanAttention show that even when the cache stays logically in the serving runtime, sequence partitioning, head-level sparsity, page selection, tile partitioning, and reduction-side softmax state determine whether decode work can be kept local enough to avoid memory traffic becoming the service bottleneck [188, 221, 231]. The comparison across these lines is useful because it separates three mechanisms often conflated under "KV optimization": reducing bytes per token, moving bytes across tiers, and avoiding bytes through sparse or tiled access. Each mechanism improves a different boundary, and each leaves a different gap in admission, eviction, or cross-worker ownership semantics.

Punica adds a multi-tenant variant of the same problem by showing that LoRA adapter state is itself a schedulable serving object [30]. Its batching and sharing policy determines whether tenant multiplexing improves utilization or merely injects new interference into the memory path. AlpaServe extends the state surface further to model-parallel deployments, where partitioned model state and multiplexing metadata become part of the runtime control loop and the system must co-schedule request admission with cross-partition sharing efficiency [126]. SGLang carries the same theme into structured programs, where RadixAttention and compressed finite-state-machine decoding turn prompt overlap and output constraints into runtime-managed state-

sharing and decoding-control opportunities [275]. DeepSpeed-FastGen adds a complementary perspective: prompt and generation phases can be fused and split dynamically to improve throughput and reduce average and tail latency for long-prompt workloads, again showing that serving policy is ultimately a policy over evolving short-lived state rather than over compute kernels alone [79].

Recent disaggregation work makes the lifecycle argument even sharper. DistServe separates prefill and decode into distinct resource pools and shows that goodput improves when the runtime treats phase-specific KV state, transfer paths, and queue imbalance as schedulable objects rather than forcing one compromise deployment across all requests [278]. Splitwise reaches a closely related conclusion through phase splitting and placement control: the bottleneck is often not how densely state is packed on one device, but when the evolving decode working set is transferred, where it resides, and how much interference that transfer injects into concurrent requests [172]. Considered alongside Sarathi-Serve and FastGen, these systems shift the design question from allocator efficiency alone to end-to-end lifecycle governance over admission, placement, transfer, and reclamation of short-lived state. Their remaining weakness is that most evaluations still assume relatively narrow request mixes and expose limited cluster-level memory-ownership or admission semantics.

Historically, these papers clarify a progression in serving infrastructure. Clipper treated serving as an orchestration layer over caching, batching, and model choice. Nexus and Clockwork moved cluster scheduling and predictability into the runtime core. InferLine and INFaaS lifted provisioning, heterogeneous replica selection, and pipeline management into the control plane. Orca exposed iteration-level scheduling pressure, while vLLM reframed memory allocation as the bottlenecked state plane. AlpaServe, Punica, DistServe, Splitwise, SGLang, and FastGen then showed that once requests become model-parallel, multi-tenant, long-context, structured, or cluster-scale, the runtime must optimize reuse boundaries, partition multiplexing, phase transitions, adapter sharing, and cross-stage ownership together rather than one layer at a time [30, 34, 35, 68, 79, 115, 126, 172, 186, 192, 234, 275, 278]. That progression is precisely why KV-cache management belongs inside a broader state-management survey rather than in a narrow inference-kernel discussion.

At the level of mechanism, these systems show that KV-cache management is a state-management problem. The cache persists across token steps, affects future scheduling decisions, exposes a reuse surface across requests, and demands explicit policies for allocation, sharing, reclamation, placement, and transfer. Once stated this way, KV-cache paging, prefix reuse, decode scheduling, and phase disaggregation sit naturally beside window management, dictionary placement, and retriever maintenance as instances of the same systems abstraction. The remaining systems gap is lifecycle closure: most serving stacks still optimize allocation, scheduling, or disaggregation in isolation and rarely model how admission, cross-node transfer, reclamation, and retrieval-side updates interact over long horizons.

Closely related work on compressed stream processing without decompression and data-aware adaptive compression extends this view to database-style execution [265, 266]. Here, state-aware execution means choosing representation, operator placement, and access path jointly. The system's real unit of optimization becomes a stateful data path rather than an isolated operator.

3.2.4 Approximation and Quality Boundaries. Approximate execution adds another layer: the system must reason not only about cost but also about output quality. PECJ addresses disorder in stream window joins by proactively compensating for missing or late data using explicit error modeling [246]. LibAMM studies approximate matrix multiplication and shows that algorithm choice, dataset properties, and memory behavior jointly determine whether approximate computing yields a stable efficiency-accuracy trade-off [243]. Adaptive sampling systems for joins and

video databases reach similar conclusions: approximation is useful only when quantity, quality, and system cost are modeled together [203, 230].

These works suggest a generalized principle for state-aware execution. When intermediate or retained state influences correctness, one must model an *execution boundary* rather than a single performance target. This boundary is defined by hardware topology, state access cost, and quality tolerance. Stable gains occur only within a limited operating region.

3.2.5 Toward Closed-Loop Execution Control. The next step is to unify these mechanisms into online control loops. Existing systems have already established many ingredients: topology-aware mapping [249], energy-aware decomposition [245], and quality-aware compensation [246]. What remains comparatively underdeveloped is an integrated runtime that combines state observability with online execution control across heterogeneous processors and service objectives. This gap is particularly important for modern inference stacks, where accelerator choice, memory transfer, approximation, and retrieval quality all interact.

3.3 State Evolution and Reuse

3.3.1 From Updates to Long-Horizon Memory. Running example C makes the evolution problem tangible: a RAG service ingesting daily document churn can keep answering queries, but quality silently drifts unless refresh cadence, shard exposure, and compaction debt are co-managed. "Update more" and "serve faster" become conflicting objectives unless the runtime prices their future interaction.

The third major dimension of state management in parallel and distributed systems concerns what happens after execution produces new information. In many older data systems, state updates were treated as maintenance. In dynamic AI services, updates are themselves central: the system must decide what to absorb, what to retain, what to discard, and how to expose retained state for future reuse.

Online adaptation and clustering systems make this shift visible. Adaptive sentiment-analysis pipelines frame model update as part of a coupled co-training loop over evolving streams, while modular stream-clustering systems expose summarization, windowing, refinement, and outlier control as interacting maintenance decisions rather than one monolithic update rule [219, 224]. The earlier empirical study of data stream clustering sharpens the same point by decomposing the design space into summarization structure, window model, outlier handling, and offline refinement, showing that update behavior and clustering quality cannot be compared meaningfully when these maintenance dimensions remain entangled [216]. Read together, these works show that online update is not a single mechanism but a coupled control problem.

3.3.2 Memory Budgets, Retention, and Sample Selection. Once updates become continuous, retention becomes equally important. Recent online continual-learning systems show that memory budgeting is not a background constraint but an active control surface: update rate, memory budget, and pipeline structure must be coordinated, and retained state must encode future representativeness rather than recency alone [121, 281]. This lesson extends beyond continual learning: dynamic benchmark efforts such as CANDOR-Bench expose how evolving vector indexes and open-world streams stress both update and retention paths at the same time [213]. In all of these cases, a runtime must preserve future reuse value while controlling the immediate cost of updates.

3.3.3 Structured Memory and Dynamic Retrieval. Recent retrieval-augmented and knowledge-enhanced systems make reuse more explicit by turning external knowledge into a managed runtime substrate rather than a static side database. Knowledge-graph-enhanced LLM frameworks organize external knowledge into reusable, queryable structures, while dynamic

retriever-adaptation systems show that without controlled memory evolution, reuse quality decays over time as corpora shift [124, 125, 259]. The state object here is not just a parameter vector or cache line but a living retrieval substrate that must be updated without destabilizing service quality. The boundary with model-centric retrieval work is important: we use retrieval and knowledge-enhanced papers in the main synthesis only when they make memory objects, planner choices, or maintenance boundaries runtime-visible. Papers that improve answer quality without exposing such control surfaces remain useful context, but they do not by themselves constitute evidence of a state-management mechanism.

Recent memory-centric systems also begin to make the middleware layer itself explicit, but the current evidence is still uneven. Neuromem argues for evaluating external memory modules under an interleaved insertion-and-retrieval protocol and decomposes their lifecycle into data structure, normalization, consolidation, query formulation, and context integration, making accuracy decay and latency cost visible as lifecycle properties rather than prompt-level artifacts [255]. SAGE points in a workflow-native direction by exposing pluggable vector-index layers, asynchronous update queues, declarative window operators, and dual-stream joins as modular stateful services for LLM-augmented reasoning workflows [138]. Placed alongside FlowRAG and Self-RAG, these lines suggest that memory middleware may eventually need to externalize both retrieval policy and memory-maintenance policy if long-horizon reasoning is to stay controllable. Their roles in the emerging stack are complementary rather than redundant: FlowRAG mainly studies how a retriever should adapt when corpus drift accumulates; Neuromem studies which parts of the memory lifecycle should be measured and varied under interleaved insert-and-query workloads; and SAGE sketches where those memory operations might live architecturally once reasoning becomes a modular, multi-stage service. As a set, they span adaptation policy, lifecycle observability, and workflow-native actuation, which is precisely why none of them alone closes the memory-middleware problem. The missing layer is a policy kernel that can use lifecycle telemetry to decide when to consolidate, refresh, defer, or expose memory objects across heterogeneous reasoning workflows rather than inside one retriever or benchmark harness.

Recent work has made clear that retrieval memory no longer behaves like a static side database, but increasingly functions as a runtime-managed layer. REALM and RAG made this shift explicit by coupling parametric generation with a retriever-facing document memory whose freshness and lookup quality directly alter downstream behavior [74, 120]. RETRO and Atlas scaled that idea further: once the external memory becomes a very large chunk store or a reusable few-shot substrate, the dominant systems question is no longer whether retrieval helps at all, but how the runtime stages, refreshes, and serves that memory without letting maintenance cost erase quality gains [14, 90]. Across scales, these systems expose the same control problem: external memory is useful only when retrieval orchestration, corpus refresh, and serving-time lookup remain coordinated.

More recent retrieval-control systems make the control surface itself explicit. Self-RAG turns retrieval into a self-reflective runtime loop in which the model decides when external memory should be consulted and when its cost or noise outweighs the expected quality gain [12]. RAPTOR exposes a different control path by organizing memory into a tree of abstractions so that retrieval becomes hierarchical routing over summary state rather than flat lookup over one index [189]. Global-planning knowledge-graph systems such as Global Planning and KELDAR add a planning layer above retrieval, showing that query decomposition, atomic retrieval order, and graph traversal policy can be treated as runtime controls over structured memory rather than as prompt-level heuristics [124, 125]. Considered together with FlowRAG, these systems indicate that modern retrieval is no longer only a question of index structure; it is a governed control loop over when, where, and at what granularity reusable memory should be traversed.

Modern retrieval pipelines also show that this substrate has multiple concrete layers. Dense Passage Retrieval shifted open-domain QA from sparse lexical lookup toward a learned dual-encoder memory in which passages are materialized as dense vectors and queried through embedding similarity, effectively turning passage collections into an actively managed vector state layer [105]. ColBERT preserved more token-level interaction while still precomputing document-side representations offline, showing that retrieval systems can keep richer matching state without paying full cross-encoder cost at query time [107]. Beneath those model-level choices, product quantization and HNSW show that index structure is itself a control surface: compression ratio, graph connectivity, and search expansion determine whether the memory layer remains affordable, fast, and updateable under scale [93, 149]. HippoRAG and RAPTOR push the memory layer further by organizing retrieval through graph and hierarchical abstractions rather than a flat nearest-neighbor space, which improves multi-hop and long-horizon reuse at the cost of more complex refresh, compaction, and stale-structure management [189, 218]. These systems differ in model architecture and retrieval objective, but from a systems angle they all externalize part of inference into a reusable index whose encoding granularity, maintenance path, and query policy directly shape latency, cost, and reuse quality.

Viewing retrieval memory as a runtime state layer also clarifies the role of vector-index mechanisms that are often treated as background implementation choices. Product quantization and HNSW expose two distinct control surfaces over reusable vector state: PQ compresses embeddings to trade fidelity for packing density and tiering efficiency, while HNSW uses graph structure to trade memory overhead for low-latency approximate traversal and incremental mutability [93, 149]. When paired with DPR- or ColBERT-style retrievers, these mechanisms determine not only lookup speed but also how frequently embeddings can be refreshed, how expensive compaction becomes, and how much update churn the serving path can absorb before quality or latency degrades [105, 107]. This is why dynamic-memory systems such as FlowRAG and CANDOR-Bench are better interpreted alongside index-maintenance mechanisms than as purely modeling papers: a substantial part of lifecycle debt sits in vector-state maintenance, not only in retriever training [213, 259].

Continuous-ANNS results make that maintenance boundary concrete. CANDOR-Bench shows that under dynamic open-world streams, insertion pressure, deletion churn, and distribution shift can move an ANN system off its recall-latency operating point long before the retriever architecture itself changes [213]. From a systems perspective, this means vector-index quality is an online state-governance problem rather than a static data-structure choice: rebuild cadence, compaction debt, and freshness policy must be evaluated on the same timeline as retriever adaptation and memory evolution [213, 259]. The broader implication is that reuse requires an intermediate layer that makes memory objects updateable, retrievable, and schedulable, and that layer is increasingly becoming the heart of intelligent-service infrastructure.

KV caches illustrate an instructive boundary case here. They are much shorter-lived than a knowledge graph or a continual-learning memory buffer, yet they still qualify as reusable runtime state because future decoding quality and cost depend on whether they are retained, shared, compacted, or discarded at the right moment. This reinforces a central point of the survey: state evolution is not limited to permanent knowledge stores, but also includes rapidly changing execution state whose lifetime is short even though its control consequences are immediate and system-wide. At that point memory stops being an auxiliary feature and becomes infrastructure, which is why services that need repeated retrieval, bounded forgetting, and controlled update latency increasingly require dedicated memory middleware with explicit admission, placement, retention, and maintenance policies.

3.3.4 Bridging Evolution with Access and Execution. A central implication is that evolution cannot be managed independently. Continuous updates create future access hotspots. Retention decisions alter memory footprint and execution cost. Retrieval quality affects whether approximate or hardware-conscious execution is acceptable downstream. Future systems should therefore treat memory evolution as an online control loop coupled to access and execution rather than as a background adaptation mechanism.

4 COMPARATIVE SYNTHESIS ACROSS DOMAINS

The preceding sections established a broad mechanism inventory across streaming, serving, retrieval, and retention, but coverage alone does not yet show which lessons actually transfer. The harder reader-facing question is therefore comparative rather than descriptive: when do these literatures solve the same control problem under different names, and when are their similarities only superficial? This section answers that question by normalizing the surveyed work to one comparison unit—state object, control surface, coupling path, evaluation boundary, and remaining gap—so that transferable systems mechanisms can be separated from domain-local optimizations that look strong in isolation but do not reliably compose across workloads and disturbance regimes.

4.1 Comparative Criteria and Integration Principles

Cross-domain synthesis is only meaningful if papers are normalized to a common comparison unit. Throughout this survey, we therefore interpret each line of work through the same analytical scaffold: the dominant state object, the runtime control surface exposed over that state, the coupling path that propagates local decisions to system-wide behavior, the evaluation boundary optimized or bounded by the paper, and the systems gap that remains afterward. This normalization matters because state-management papers often report improvements under incompatible metrics or horizons. Without a common comparison unit, a survey quickly degenerates into a chronology of local wins rather than a cumulative account of reusable systems mechanisms.

Using that scaffold, we group papers by mechanism family rather than by venue order or application label alone. The key comparison question is not simply which domain a paper belongs to, but whether it advances migration control, memory reclamation, phase-aware scheduling, retrieval-index maintenance, approximation-aware execution, or another recurring control problem. We treat a subsection as analytically mature only when it moves beyond enumeration and contrasts multiple mechanisms under a shared service boundary. That is why the more mature subsections in this manuscript end not only with literature coverage, but also with an explicit unresolved control seam, design implication, or evaluation gap.

To keep the survey cumulative rather than encyclopedic, we apply three integration checks throughout the manuscript: terminology must remain consistent across adjacent clusters, citations must resolve to clearly comparable mechanisms rather than nearby but mismatched lines of work, and each paragraph must connect mechanism to boundary rather than report contributions in isolation. These criteria are stricter than a simple completeness goal, because the article should not merely list what has been built, but explain which abstractions generalize, which assumptions fail to transfer, and which open systems questions remain stable across domains. They also imply an explicit evidence hierarchy: a paper that makes a control seam visible is not yet equivalent to one that shows the seam remains governable under disturbance, multi-tenancy, and lifecycle interaction.

4.2 Cross-Domain Comparative Synthesis

The following sections provide domain-specific comparative syntheses. Each subsection answers the same three questions: which state object dominates in the domain, which runtime controls

Table 4. Reader guide for the cross-domain synthesis.

Cluster	Dominant state object	Reader question	Easy comparison mistake
Streaming and transactional dataflow	windows, progress metadata, ownership logs	When is state safe to expose, move, or replay?	separating locality, migration, and recovery into unrelated mechanisms
Hardware-conscious and approximate execution	encoded data paths, summaries, quality-sensitive intermediates	Which representation and movement choices preserve the service boundary?	reading kernel or device speedup as an end-to-end win by default
LLM serving	KV pages, prefix state, adapter or model residency	Which short-lived objects may be admitted, reused, transferred, or restored without violating the request boundary?	reducing the problem to allocator efficiency alone
Retrieval and retention	indexes, planner-visible memory, replay buffers, budget traces	How much lifecycle debt can the runtime tolerate before quality collapses?	conflating invocation policy with maintenance, publication, or exposure policy

are mature, and which control seams remain under-developed. Keeping that triad fixed matters because it prevents the discussion from collapsing back into domain-local storytelling: the goal is not only to summarize each area faithfully, but also to surface where apparently different systems are converging on the same governance problem under different names and workload assumptions.

Table 4 is meant as a compact reading key for the dense discussion that follows. The point of the next four clusters is not to decide which domain is most mature in the abstract, but to track where the same governance question reappears under different state lifetimes, maintenance costs, and service boundaries.

4.2.1 Streaming and Transactional Dataflow Systems. Streaming and transactional dataflow systems are the clearest historical example of state management becoming a runtime control problem rather than a storage afterthought. Their shared challenge is not simply to keep state correct, but to keep it continuously updateable while preserving three boundaries at once: when state becomes logically visible, where it is currently owned, and how it can be replayed or moved under disturbance. The most useful way to compare this literature is therefore by mechanism family rather than by engine generation. Across the papers reviewed here, progress comes from making one of these boundaries explicit and then turning it into a runtime control surface.

The first mechanism family is *visibility and progress semantics*. Aurora and Borealis already treated operator graphs and adaptation paths as runtime-managed artifacts, but later systems made the visibility boundary itself first-class. MillWheel, Naiad, and the Dataflow model elevated event time, pending work, watermarks, triggers, and partial-order progress into explicit runtime state rather than hidden bookkeeping [1, 2, 7, 8, 157]. Their common contribution is to separate two decisions that older systems often conflated: when an update is externally meaningful, and when the underlying state is stable enough for buffering, replay, checkpointing, or reordering. This distinction is foundational because it changes the unit of control from operator throughput to controlled state advancement. The tradeoff is equally clear: richer progress semantics improve

correctness and flexibility, but they also introduce metadata, coordination, and reasoning overhead that later control paths must absorb.

The second family is *safe advancement over partially materialized state*. Trill and Differential Dataflow show that incremental maintenance is not only an algebraic optimization; timestamps, differences, and frontier metadata determine which portions of shared state may advance without globally stalling the computation [23, 153]. Flink and StreamCloud expose a more deployment-oriented variant of the same seam: elasticity and state movement are viable only when the runtime has a concrete notion of state ownership and transfer, rather than treating operator state as opaque memory local to one execution context [20, 69]. Viewed comparatively, these systems reveal a distinction that is often lost in paper-by-paper summaries: progress metadata answers when a transition is safe, whereas ownership-transfer mechanisms answer where that transition may occur. Effective runtime management therefore requires both. Systems with strong logical progress but weak movement semantics struggle under reconfiguration; systems with movable state but coarse progress semantics pay coordination penalties or expose fragile correctness envelopes.

The third family is *topology- and conflict-aware access control*. Once state is long-lived and shared, locality and contention cease to be passive workload properties; they become endogenous consequences of the scheduler itself. Revisiting multicore stream processing and multi-query optimization for complex event processing exposed how hidden queues, synchronization surfaces, and shared subplans can dominate scalability even when operator logic is simple [256, 261]. BriskStream, concurrent stateful stream processing, and stateful streaming joins sharpen that lesson by showing that hotspot placement, lock granularity, NUMA distance, and index-aware synchronization jointly define the access-cost surface [252, 257, 262]. The comparative point is important: sharing work is not automatically beneficial. Under skew or ordering constraints, aggressive sharing can amplify interference faster than it saves computation. What distinguishes the stronger systems in this family is that they treat observability, partitioning, and synchronization as one coupled control problem rather than three isolated optimizations.

These access decisions already encode a latent disturbance policy. A placement that concentrates hot keys onto a small set of workers may raise steady-state throughput, but it also determines where barriers accumulate, which partitions replay together, and how much migration debt appears during reconfiguration. This is why locality work and recovery work are best interpreted jointly rather than as unrelated subtopics [19, 150, 257, 262, 270]. More specifically, access topology is never only a performance choice; it is also a commitment about future recovery shape.

The fourth family is *ownership transfer and recovery coupling*. Asynchronous checkpointing in Flink showed early that fault tolerance overhead depends on how consistently the runtime aligns barriers, in-flight records, and snapshot boundaries with normal execution [19]. Operator-state-management work, Megaphone, and Spacker extend that line by turning scale-out and rebalancing into explicit state-transfer mechanisms rather than administrative afterthoughts [51, 150, 160]. Transactional stream systems such as MorphStream and recent fast-recovery designs add a tighter dependency-aware perspective: replay scheduling, recovery metadata, and steady-state execution plans must be co-designed because dependency shape determines both throughput stability and post-failure convergence [151, 270]. The major cross-paper tradeoff is between transfer granularity and control complexity. Fine-grained migration and dependency-aware recovery reduce visible pause time, but they increase metadata pressure and require stronger ownership semantics if the system is to remain auditable under disturbance.

Read together, these lines imply a latent ownership-transfer contract: state must move with explicit progress fences, bounded dual-ownership windows, and replay idempotence so the same shard can be handed off during scale-out and after failure without redefining correctness [19, 51,

150, 160, 270]. What remains missing is a reusable specification of this contract that can be applied across migration, elasticity, and recovery rather than re-derived per subsystem.

Neighboring database and replicated-log systems indicate what such a specification would likely need to contain. Spanner contributes leaseholder continuity, Calvin contributes deterministic replay order, Raft contributes term-scoped writer authority, and RAMCloud contributes fast reconstruction of serving ownership after failure [33, 164, 165, 204]. None of these systems solves streaming handoff directly, but together they clarify that ownership transfer is not only a data-movement event. It is also a protocol event over writer legitimacy, replay scope, and old-owner retirement. That is precisely the part still under-specified in most stateful stream runtimes.

The arc of this literature shows a clear maturation path. Early systems made state visible; mid-generation systems made it advancable and movable; newer systems try to make it recoverable and reschedulable without duplicating control logic across steady-state, elasticity, and failure handling. What remains missing is a unified disturbance-aware controller that spans all three boundaries simultaneously. Most engines still optimize one seam at a time: progress semantics under relatively stable placement, locality under limited fault modeling, or recovery under simplified tenant and workload structure.

That gap becomes more serious under modern deployment assumptions. Multi-tenant interference, rapid skew flips, and state objects with semantic structure all break the older assumption that conflicts are mostly per-key and that one-off reconfiguration experiments approximate long-running operation. Even in transactional streaming, many evaluations still separate throughput, migration, and recovery into distinct scenarios instead of measuring them as interacting phases of one control loop. The central open systems question is therefore not whether streaming runtimes can manage state efficiently in one mode, but whether they can unify visibility, ownership, and disturbance response under one reusable state-governance contract. Until that happens, streaming and transactional dataflow will remain a rich source of mechanisms, but not yet a closed solution to end-to-end state management.

4.2.2 Hardware-Conscious and Approximation-Aware Stateful Execution. Hardware-conscious systems established that stateful efficiency depends on movement and representation choices as much as arithmetic intensity. Fine-grained CPU/GPU coordination, dictionary-aware processing, and compression-aware scheduling illustrate this shift [245, 249]. The domain's strongest contributions expose state representation as a runtime control surface rather than a static storage decision.

That control surface is broader than device placement alone. CompressStreamDB and adaptive compression systems show that the runtime can often avoid needless decompression or memory movement by changing representation format, operator path, and compression level together rather than optimizing each stage independently [265, 266]. More specifically, the optimized object is not merely a kernel but a stateful data path whose fidelity, placement, and transfer shape the attainable service envelope. Figure 4 summarizes this seam structure: representation decisions, execution-path choices, and quality boundaries are coupled controls rather than separate subsystems.

Approximation-aware systems add a boundary model that is still underused outside their subcommunity. BlinkDB, ApproxHadoop, ApproxJoin, StreamApprox, and IncApprox already made the essential systems point years ago: once a runtime saves work by operating over samples or summaries, it must control not only response time but also update debt, reuse value, and error stability over time [5, 31, 64, 82, 214]. More recent mechanisms such as PECJ, FreeSAM, LEAP, and LibAMM push that logic closer to online control. They show that proactive compensation, adaptive sample sizing, predictive sample maintenance, and approximate kernel selection are valuable precisely because they couple quality boundaries to runtime cost decisions [203, 230, 243, 246].

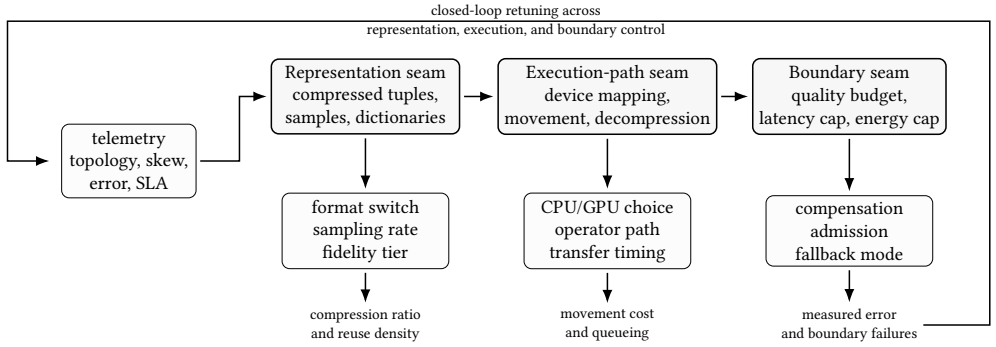

Fig. 4. Control seams in hardware-conscious and approximate execution. Representation choices, execution-path decisions, and quality-boundary enforcement form one closed-loop control problem over stateful data paths.

Without this coupling, approximation can improve local throughput while degrading user-level utility.

Within this synthesis, three mechanism families recur. The first is *state-path shaping*, where FineStream, CStream, CompressStreamDB, and adaptive compression treat movement, encoding, and device mapping as one execution surface [245, 249, 265, 266]. The second is *budgeted approximation*, where BlinkDB, ApproxHadoop, ApproxJoin, StreamApprox, and IncApprox treat sample or synopsis state as a reusable asset whose maintenance policy defines the real accuracy-latency frontier [5, 31, 64, 82, 214]. The third is *online boundary steering*, where PECJ, FreeSAM, LEAP, and LibAMM make compensation, sample adaptation, predictive materialization, or kernel choice part of the runtime loop itself [203, 230, 243, 246]. Their common systems lesson is that quality budgets must be first-class scheduling inputs, not post-hoc validation checks.

The comparison among these three families is important because they intervene at different boundaries. State-path shaping changes how existing state moves and is represented under a mostly fixed correctness target; budgeted approximation changes how much state fidelity the service can afford to retain; online boundary steering changes when the runtime is allowed to spend quality budget at all. A mature execution runtime will likely need all three layers rather than one substitute for the others.

Recent hardware-conscious work broadens the state object from compressed tuples to the entire residency fabric, but the comparison is clearest when split by control problem. One line studies *tiering and residency governance*. HMG, G10, MLP-Driven Offload, Colloid, Beyond Hotness, MemStrata, FVM, HyCache, DRAM-Cache, TierTune, and ZICO all expose tensor pages, cachelines, feature maps, or multi-tier page state as actively managed objects whose migration, prefetch, and reclamation policy determines whether extra memory capacity reduces stalls or merely shifts them across links and tiers [80, 94, 114, 118, 131, 137, 152, 184, 206, 253, 279]. These systems differ in substrate, but they share one coupling path: movement debt accumulates whenever the runtime expands capacity without also governing which state must remain close to compute.

A second line studies *metadata and ownership overhead* rather than raw capacity. PFSCK, PolarDB, CXL-enabled end-host designs, ToLeO, CLAP, and MCU-oriented lifecycle coordinators show that once memory is disaggregated, persistent, or spread across chiplets and devices, consistency metadata, freshness versions, mapping locality, and buffer ownership become runtime costs in their own right [18, 40, 41, 95, 170, 276]. Jin et al.'s concealing scheduler reaches a similar conclusion from the I/O path: compression state is only useful when the runtime can place it into

barrier slack without extending the critical path, which means even offline-looking I/O reduction is really a scheduling problem over future movement debt [100]. Placed beside the tiering work above, the stronger lesson is that capacity expansion is never free; it creates a second governance problem over movement debt, metadata locality, and failure-visible ownership.

The accelerator and heterogeneous-inference literature pushes the same point into *execution-path specialization*. One subgroup makes representation and decompression part of the control loop: FuseMax, AutoScratch, Coruscant, Anda, DECA, QFactory, SampleAttention, FlashInfer, ATOM, DecDec, QuantLLM, and SPInfer show that streaming softmax statistics, scratchpad-managed cachelines, sparse weight formats, activation bit-planes, decompression metadata, quantization tiles, and sampled attention summaries must be transformed carefully enough that bandwidth savings survive the decode path [45, 46, 53, 61, 101, 161, 171, 225, 233, 254, 273, 283]. A second subgroup studies heterogeneous placement across devices or near-memory engines: Clone, Hermes, CENT, PowerInfer, PIMBA, HeteroInfer, LIA, Mercury, GMLake, WaferLLM, TurboAttention, DIP, SAVE, Duplex, PodAttention, FlexInfer, PAISE, and SmartInfinity decide which model, KV, optimizer, or phase-specific state should stay on GPU, shift to CPU or PIM, or move nearer to storage [29, 48, 66, 67, 70, 76, 92, 103, 104, 108, 110, 117, 139, 159, 197, 205, 240, 277]. Taken together, these systems show that acceleration is now a runtime question of how state remains compressed, local, sparse, offloaded, sampled, or phase-specialized without breaking the end-to-end service boundary. Their common limitation is portability: most assume one hardware contract, one workload regime, or one phase split, so they stop short of an ownership model that survives cross-device migration, multi-tenant interference, and long-horizon lifecycle control.

This comparative synthesis suggests two stronger cross-domain lessons. First, state representation should be treated as a dynamic control variable rather than a compile-time parameter: compression level, summary fidelity, and decompression path all reshape execution cost surfaces in the same way that batching or page layout reshapes serving cost surfaces. Second, error budgets should be compared as service-boundary contracts rather than as local approximation settings. A runtime that knows how to trade error for speed but does not know when that trade breaks p99 latency, downstream join quality, or future update cost is still operating open-loop.

Open gaps include weak portability of policy tuning across hardware topologies, limited composition between compressed-state execution and approximation policies, and very weak integration between approximation control and recovery semantics. A system may know how to bound error during normal operation yet still fail to define what error contract survives reconfiguration, replay, or cross-device migration. That missing recovery-time boundary model is the clearest sign that this part of the literature still lacks a fully closed-loop runtime.

At this point, a broader comparative pattern should be visible. Streaming centered on visibility and ownership, hardware-conscious execution centered on movement and quality boundaries, and the next cluster shifts the focus again toward short-horizon lifecycle control under severe memory pressure. LLM serving matters in this sequence not because it is fashionable, but because it compresses many older state-management tensions into one runtime where admission, reuse, transfer, and restoration all become visible within milliseconds.

4.2.3 LLM Serving and Short-Lived Memory Lifecycles. LLM serving is the clearest current example of state management becoming the runtime rather than merely supporting it. In many deployments, KV caches, prefix-sharing structures, phase-skewed queues, and multiplexed model-state layers define the real service envelope more directly than the model graph alone [6, 30, 115, 126]. The most useful way to read this literature is therefore not as a race for the fastest inference

stack, but as a sequence of systems that make previously hidden memory interference visible to scheduling, admission, and ownership control.

Under that lens, the serving literature is most comparable when read as four control seams over one short-lived state lifecycle. Clipper and Clockwork externalize orchestration and predictability, turning batching, caching, and dispatch timing into explicit SLO-facing controls [35, 68]. Paging-style runtimes externalize allocation and sharing over KV state, thereby widening the feasible batching region through memory-layout policy rather than through model changes alone [115]. Sarathi, DistServe, and Splitwise externalize phase asymmetry, making prefill and decode schedulable as distinct resource regimes instead of one blended request path [6, 172, 278]. Nexus, InferLine, and INFaaS operate mainly at the service and provisioning boundary, whereas FlexGen, Punica, and AlpaServe move deeper into memory placement, tenant multiplexing, and model-partition residency inside the execution substrate [30, 34, 126, 186, 192, 195]. The synthesis point is therefore narrower than a claim of identical evaluation targets: each family makes a previously hidden source of interference schedulable, but none by itself explains how request-level admission, phase-level scheduling, and memory-residency control should compose under mixed tenants and mixed phases.

Recent follow-on systems sharpen that fourth seam by splitting the same problem across three governance levels. S-LoRA treats adapter weights and KV tensors as one multiplexed paging domain, so the question is whether parameter-efficient tenants can share a base model without paying hidden fetch and batch-fragmentation costs [193]. Jenga moves one layer inward and shows that even within one model family, heterogeneous cache shapes make allocator structure and layer-aware eviction part of the serving control surface rather than background memory bookkeeping [248]. Prism moves one layer outward to the multi-model regime, where KV memory, model residency, and reactivation delay must be co-governed because bursty demand otherwise strands memory in idle models while hot models miss their SLOs [236]. Set beside Punica and AlpaServe, the synthesis point is sharper than "better batching": modern serving needs a transferable notion of ownership and reclaimability that survives across adapters, heterogeneous cache layouts, and whole-model residency under one control loop.

Adapter and multi-model systems further show that the active state pool is no longer homogeneous. D-LoRA and Chameleon treat adapter residency, eviction, and request-adapter co-placement as online scheduling surfaces, exposing a conflict between cache warmth, fair queueing, and cross-replica load balance [88, 223]. Toppings adds a heterogeneous execution angle: adapter rank determines whether CPU-assisted computation or GPU-resident locality is the right control response, so the scheduler must reason about adapter shape rather than only adapter identity [123]. Aegaeon extends the ownership question to pooled deployments, while PlanetServe suggests how the same question may reappear in decentralized overlays, where model weights, heterogeneous KV state, session affinity, routing metadata, and even trust or verification state influence whether reuse can be obtained without centralizing all control [47, 226]. Across these systems, the open problem is not merely scaling to more tenants; it is defining ownership, lease, and invalidation semantics for state objects whose reuse value crosses request, model, adapter, and node boundaries.

The same seam becomes more complex at cluster scale. DistServe, Splitwise, and FastGen show that separating prefill from decode is useful precisely because the two phases stress different resources, but Punica and AlpaServe make clear that the state being routed extends beyond generic KV memory to include tenant-specific adapters, partition metadata, and multiplexing constraints whose residency rules change interference patterns [30, 79, 126, 172, 278]. Taken as complementary evidence across different control horizons, these systems and earlier service-level planners such as Nexus, InferLine, and INFaaS indicate that short-lived serving state already lives under a

cluster-level governance loop of occupancy signals, queue backpressure, provisioning decisions, and admission control [34, 186, 192]. Future runtimes should therefore treat phase splitting and tenant multiplexing as a joint optimization problem, while still keeping explicit which decisions protect p99 latency, which protect utilization, and which protect placement stability under tenant churn.

Recent follow-on work suggests that phase disaggregation creates a second-stage bottleneck: once prefill and decode are separated successfully, transfer and restoration of KV state become control surfaces in their own right. Prefill-as-a-Service argues that next-generation hybrid-attention models may make cross-datacenter prefill offload increasingly plausible, but only when selective offloading, bandwidth-aware scheduling, and cache-aware placement are co-designed rather than treated as deployment afterthoughts [177]. SplitZip and CacheFlow sharpen the same point from the data path itself. SplitZip treats GPU-friendly lossless KV compression as an online transfer optimization for disaggregated serving, while CacheFlow frames KV restoration as a batch-aware multi-dimensional scheduling problem across tokens, layers, and GPUs [73, 162]. Combined with DistServe and Splitwise, these systems suggest an emerging serving frontier in which the key question is no longer only where phase boundaries should be drawn, but how transfer, restoration, and cross-cluster placement should be governed as one short-lived state lifecycle.

Several recent disaggregation and heterogeneous-serving systems make parts of that lifecycle explicitly predictive rather than purely reactive. WindServe, GLLM, Past-Future, HETIS, Seesaw, and ThunderServe each treat some future-facing state signal as schedulable: prompt-length skew and decode-load imbalance, prefill/decode token allocation, predicted output-length memory peaks, head-local KV residency across heterogeneous GPUs, phase-specific parallelism layouts, or heterogeneous cloud service-group placement [49, 65, 72, 98, 155, 198]. Collectively, they indicate a shift from reactive memory reclamation toward anticipatory governance over future cache occupancy, phase imbalance, and transfer debt, rather than a settled predictive-serving architecture. Their shared limitation is also clear: most rely on profiled or sampled workload structure and still lack a portable contract for what happens when predictions are wrong, when pool sizes must change online, or when cross-cluster ownership and isolation constraints dominate the predicted throughput gain.

That comparison becomes more precise when these systems are grouped by the debt they try to predict. WindServe and GLLM mainly forecast phase-imbalance debt: prompt-length skew or token-allocation mismatch matters because it lets the runtime rebalance prefill and decode before queue asymmetry hardens into migration or bubble cost [49, 72]. Past-Future instead forecasts memory-pressure debt by estimating future output length and KV occupancy peaks, then converting that forecast into admission headroom before continuous batching walks into avoidable eviction and SLA loss [65]. HETIS, Seesaw, and ThunderServe focus on heterogeneity debt, using head-local KV placement, phase-specific parallelism changes, or heterogeneous service-group assignment to decide which nonuniform hardware path should inherit the next phase of state [98, 155, 198]. Read together, these papers are better viewed as a family of early anticipatory controllers over queue growth, cache pressure, and transfer asymmetry than as one mature predictive-serving design line. The remaining gap is uncertainty composition: practical runtimes still lack a portable rule for reconciling wrong or conflicting predictions about load, memory, and placement before those predictions start issuing incompatible admission, migration, or reshaping actions.

Prediction alone, however, does not explain when state can be reused safely. Another mechanism family therefore makes reuse structure explicit rather than merely shrinking footprints. SpecInfer, FastTree, InstAttention, PromptCache, Marconi, ICCache, CacheBlend, DroidSpeak, CacheSlide, and PIE all exploit reusable prefix or branch state, but they expose different control surfaces: speculative tree growth, shared-context query grouping,

modular prompt segmentation, hybrid-state admission, chunk-level hash indexing, RAG-side cache fusion, cross-model transfer, positional correction, or application-visible KvPage APIs [62, 63, 142, 143, 154, 166, 167, 169, 232, 238]. Viewed jointly, these papers show that reuse is not a single cache-hit question. It depends on whether the runtime can describe semantic compatibility, branch validity, and invalidation scope precisely enough to avoid re-prefill without accidentally reusing the wrong state.

A separate but adjacent family extends the lifecycle in time rather than in structure. Pensieve, CachedAttention, HCache, InfiniGen, Mooncake, Prefill-Only, LayerKV, FastServe, JITServe, token-fair serving, LLMnix, MELL, Weaver, Libra, Symphony, HydraServe, and ServerlessLLM collectively frame session history, tiered cache placement, restoration policy, preemption state, fairness counters, live migration, and cold-start model residency as coupled governance problems rather than isolated cache policies [4, 43, 55, 56, 58, 59, 119, 140, 146, 178, 187, 194, 199, 222, 227, 235, 264]. VAttention and KV Cache in the Wild sharpen the same comparison from the allocator side by showing that address-space structure and real reuse distributions matter as much as policy cleverness, while MoE-Lightning demonstrates that sparse-model serving inherits the same short-lived lifecycle tension for expert and offloaded state [17, 176, 209]. The resulting systems lesson is broader than "better caching": once requests survive across turns, tiers, warm starts, or elastic pools, goodput depends on coordinating eviction, prefetch, migration, restoration, and fairness against explicit service debt such as TTFT, preemption loss, and reactivation delay.

The comparison becomes clearer when this family is split by where in the lifecycle the runtime intervenes. Pensieve, CachedAttention, HCache, Mooncake, and Symphony mainly govern the session-restoration boundary: they assume conversational or multi-turn state already has reuse value, then differ on whether that value is preserved through local swap scheduling, scheduler-visible prefetch, hidden-state persistence, cluster-wide remote cache reuse, or advisory-triggered asynchronous restore [4, 56, 58, 178, 235]. FastServe, token-fair serving, LLMnix, MELL, and Libra intervene at the active-contention boundary, where the issue is how preemption, live migration, fair token allocation, and split-position control distribute latency debt once many requests compete for the same memory pool [140, 187, 194, 199, 222]. HydraServe, ServerlessLLM, LayerKV, and Prefill-Only operate earlier at the bootstrap-residency boundary, showing that cold-start weight fetch, tiered checkpoint loading, early-layer KV retention, and prefill-only specialization already determine which state can ever become reusable before steady decode begins [43, 55, 146, 227]. Read together with VAttention, KV Cache in the Wild, and MoE-Lightning, the lesson is sharper than a generic call for better caching: temporal-lifecycle control works only when the runtime can align allocator realism, reuse distribution, and service debt across restore, migrate, fair-share, and startup decisions [17, 176, 209]. The remaining gap is a unified controller that can compare those actions on the same TTFT/TPOT/SLO boundary instead of letting each sub-policy optimize a different phase in isolation.

This contrast sharpens the local tradeoff structure and also clarifies why many results still feel hard to compare. Predictive serving tries to avoid future debt by forecasting load or memory pressure early; structural-reuse systems try to avoid redoing work by making validity and compatibility explicit; temporal-lifecycle systems accept that debt will recur and instead optimize how restoration, migration, and fairness are paid over time. The open design challenge is that current runtimes rarely combine these three views into one policy loop, so semantic reuse, predictive admission, and long-horizon restoration are still evaluated mostly in isolation. Methodologically, this convergence matters as much as any single optimization result: paging and sharing work expose allocator boundaries, phase-aware schedulers expose temporal asymmetry, disaggregation work exposes transfer and ownership costs, and tenant-aware systems expose interference and admission semantics [6, 30, 73, 115, 126, 162, 172, 177, 278]. Read together, these families point

to a broader claim than "better buffering": short-lived serving memory should be evaluated as a governed lifecycle with explicit ownership, exposure, and backpressure semantics rather than as a passive buffer pool.

That lifecycle also has a correctness side that serving papers still expose only unevenly. Reusing or transferring KV state is not automatically legal just because it is fast: prefix compatibility, position or phase alignment, adapter identity, decode-branch validity, and restoration ordering all constrain whether a cache fragment may be shared, migrated, or replayed without changing the effective service path for that request. In that sense, LLM serving is converging toward an older distributed-systems problem under a new state object. The runtime needs not only a cost model for who should own short-lived state next, but also an exposure model for when that ownership change remains compatible with the request's execution semantics. This is precisely where the literature remains thinner than in mature stream or transaction systems: many papers optimize reuse, transfer, or restoration cost aggressively, but far fewer make the legality conditions of those actions explicit enough to compare across continuous batching, speculative branches, adapter multiplexing, and disaggregated decode paths.

Structured-program serving pushes this lifecycle view one step further by introducing runtime-managed reuse boundaries across multi-call workflows, constrained decoding, and retrieval subcalls [275]. In these settings, state lifecycle is no longer per-request only; it includes reusable intermediate objects whose value depends on program structure, so the control surface expands from page residency and batch formation to workflow-level admission, invalidation, and reuse scope. Read alongside Parrot, the shift is also about what the serving runtime is allowed to observe and optimize: SGLang emphasizes reusable prefix state and decoding structure inside one language-program abstraction, whereas Parrot elevates request dependencies, shared prompt structure, and end-to-end application objectives into first-class scheduling inputs through semantic variables and just-in-time DAG analysis [132, 275]. The broader mechanism is not unique to LLMs, as SONIC and ORION already showed in chained serverless applications where dependency-local data passing and warm-start requirements shift optimization from isolated function latency to end-to-end workflow completion [147, 148]. What changes in LLM serving is the state object itself: instead of only function inputs and warm containers, the runtime must govern reusable KV contexts, prompt-structure metadata, intermediate outputs, and objective-specific scheduling hints across calls. This leaves several open gaps that are architectural rather than cosmetic, including cross-node lifecycle coordination for short-lived state, global admission policies that anticipate cross-phase transfer and tenant interference, disturbance-aware benchmark protocols (burst, warmup, drift), and principled integration between serving-memory control and retrieval-memory updates.

4.2.4 Retrieval Memory and Vector-Index State Layers. Retrieval systems are easiest to misread as a sequence of better retrievers, but the more durable systems lesson is about memory governance. Once retrieval moved from lexical inverted indexes to dense and hybrid memory substrates, indexing, refresh, exposure, and serving became tightly coupled state problems rather than offline preprocessing details [74, 105, 107, 120]. The key architectural transition is therefore from "querying documents" to "maintaining a living memory layer" whose update and compaction debt must remain visible to the runtime.

Read this way, the retrieval literature is easiest to compare as three coupled control layers. External-memory orchestrators such as REALM, RAG, RETRO, and Atlas decide when indexed memory is consulted and refreshed, making corpus reuse and lookup boundaries part of the service path rather than offline preprocessing [14, 74, 90, 120]. Index-maintenance systems decide how that memory stays live under scale and churn: dense dual encoders and late-interaction retrievers shape

the online-offline split, PQ and HNSW expose compression and graph topology as runtime controls, and FreshDiskANN, SPFresh, IP-DiskANN, plus MARCO expose mutation locality, deletion handling, and disk layout as policies over future freshness debt rather than static data-structure choices [93, 105, 107, 149, 196, 228, 229, 239]. Structured long-horizon memory systems such as HippoRAG and RAPTOR alter the memory object itself by adding graph or hierarchical structure, which improves multi-hop reuse only by introducing new maintenance obligations in graph rewiring, abstraction refresh, and stale-structure cleanup [189, 218]. The common comparison point is therefore not which model family a paper uses, but which layer makes lifecycle debt explicit and which layer still hides it.

A fourth family is planner-mediated index governance. Self-RAG makes retrieval invocation itself a runtime control decision rather than a fixed pipeline stage, but that decision only remains meaningful when the runtime also understands whether the underlying memory layer is stale, fragmented, or temporarily expensive to refresh [12]. FlowRAG and CANDOR-Bench expose the next step in that control loop: once corpora and ANN structures evolve continuously, the dominant tradeoff is no longer simply retrieve versus do not retrieve, but whether to query a stale index, pay immediate rebuild or compaction cost, or defer maintenance and absorb future quality drift [213, 259]. The comparison to PQ, HNSW, FreshDiskANN, and SPFresh is therefore interface-level rather than metric-level: the former line decides when stale or expensive memory may still be queried, while the latter lines decide how compression, graph maintenance, and local repair change the cost of keeping that memory queryable at all [93, 149, 196, 229].

Read together, these papers imply a trigger hierarchy. Self-RAG decides whether external memory should be consulted at request time [12]. FlowRAG decides when evolving retriever state should be refreshed before reuse quality decays too far under corpus drift [259]. CANDOR-Bench exposes when insertion churn, deletion debt, and distribution shift have already pushed an ANN structure away from its intended recall-latency region [213]. FreshDiskANN then shows when local graph repair and disk-aware update paths are sufficient, and when deferred repair is merely accumulating future rebuild debt [196]. The systems lesson is that invocation policy and maintenance policy are not independent knobs: a planner that keeps querying stale-but-local memory may preserve short-run latency while silently worsening future answer quality, whereas an overly aggressive rebuild trigger can protect freshness but destroy end-to-end service stability by injecting maintenance cost onto the critical path.

This comparison sharpens a systems distinction that is easy to blur in RAG-centric writing. PQ and HNSW mainly optimize steady-state query cost for a chosen representation, whereas FreshDiskANN, SPFresh, IP-DiskANN, and CANDOR-Bench optimize or expose the cost of keeping that representation live under inserts, deletions, and tiered storage. Once that distinction is explicit, retrieval maintenance stops looking like background ingestion plumbing: micro-batch size, local graph repair, in-place mutation, tombstones versus physical deletion, and SSD-resident placement all become runtime controls that trade present freshness against future rebuild debt. The resulting control problem is structurally close to KV-state management in serving, but the service boundary here is recall drift under corpus churn rather than TTFT.

A complementary systems line shows that these choices do not disappear inside production vector stores. Milvus and Manu treat segment layout, shard placement, background indexing, and concurrency control as first-class data-management concerns rather than as wrappers around a single ANN structure [71, 211]. Considered alongside FreshDiskANN-style and SPFresh-style maintenance, they suggest that retrieval governance must span two coupled planes: local mutation policy inside each index shard, and cluster-level decisions about when segments are sealed, compacted, migrated, or queried under concurrent writes. The comparison is at governance scope rather than at one shared metric: ANN-maintenance papers primarily expose freshness-versus-maintenance

tradeoffs inside an index, whereas vector-store papers expose when partially refreshed state may be published safely at the service boundary.

Recent systems broaden the runtime surface around the index itself. HeteRAG, database-native vector layers such as SingleStore, Rummy, and composable-memory vector-search engines each expose a different actuation point: heterogeneous retriever routing, transactional vector-state management inside a broader data system, DRAM-disk caching and prefetch for vector pages, or memory-fabric-aware placement for large search structures [28, 136, 180, 267]. Their common contribution is to move retrieval closer to a governed data service rather than a standalone ANN library, which makes index freshness, transaction boundaries, and placement policy visible to the same runtime loop.

What makes this cluster worth separating is that the four systems intervene at different boundaries of the same retrieval lifecycle rather than offering interchangeable acceleration techniques. HeteRAG changes the service-routing boundary: once retrieval and generation live on different memory and bandwidth regimes, the runtime must decide which retriever path is worth invoking for a given request and when heterogeneous routing actually pays for its coordination overhead [136]. SingleStore shifts the control problem to the publication boundary by making vector segments, delete bitmaps, and Top()-aware planning visible to the database optimizer, so partially refreshed vector state becomes governed by transactional exposure rules rather than by best-effort ANN heuristics [28]. Rummy focuses instead on the residency boundary, where balanced IVF clusters, reordered transfer, and GPU-side grouping determine whether vector search cost appears as exposed PCIe stall or as cache-aware overlap inside the query path [267]. Composable-memory search pushes one layer further out to the fabric boundary, preserving exact-search quality by moving very large embedding state closer to the memory fabric rather than by weakening the retrieval objective itself [180]. Read together, these papers show that modern retrieval infrastructure spans routing, publication, residency, and placement as distinct control surfaces. The missing layer is a compositional trigger policy that can decide when a request should tolerate stale but local state, when publication should wait for maintenance to finish, and when page or fabric movement is justified by downstream answer-quality gain rather than raw retrieval throughput.

That cross-layer view also clarifies why retrieval systems have been hard to evaluate with one metric family. HeteRAG changes whether retrieval and generation traffic meet at the same memory tier [136]. SingleStore changes when partially rebuilt vector state becomes visible to queries and how optimizer-visible search plans constrain that exposure [28]. Rummy changes whether residency and transfer order keep vector pages local enough for GPU execution to hide movement cost [267]. Composable-memory search changes whether exact retrieval should be preserved by moving large embedding state toward the memory fabric rather than by relaxing the retrieval objective [180]. One request can therefore traverse four trigger points: invocation, maintenance, publication, and placement. A useful retrieval runtime must specify not only each local policy, but also their precedence when they disagree. Otherwise one controller may accelerate stale state exposure while another tries to delay publication or move pages, producing oscillation instead of stable freshness-quality control.

The central insight is that retrieval quality is a lifecycle property, not just a model property. Refresh cadence, planner-triggered retrieval policy, index maintenance policy, compression choice, stale-memory eviction, and continuous-ANNS maintenance under churn govern long-term quality-cost balance as strongly as encoder architecture [12, 213, 259]. The central comparative point is therefore not simply that retrieval systems use different memory objects, but that they externalize lifecycle debt at different trigger layers and under different service boundaries. Planner-mediated retrieval makes staleness visible at request time; ANN-maintenance systems make rebuild and deletion debt visible at index time; vector stores make sealing, compaction,

and shard exposure visible at publication time; placement systems make page movement and memory-fabric locality visible at execution time [12, 71, 180, 196, 211, 213, 228, 229, 259, 267]. A comparative synthesis should therefore compare these works by whether they externalize lifecycle debt into an auditable control surface, while keeping explicit whether the reported gain is request-level quality, index-level maintenance stability, cluster-level exposure safety, or placement-level latency-quality efficiency.

Seen from that angle, the remaining gaps are not isolated implementation chores but missing governance semantics across layers. The literature still lacks a unified treatment of dense, compressed, graph, hierarchical, and planner-mediated memory objects in one control plane, explicit trigger policies for when planners should tolerate staleness versus request rebuilds, deletion semantics and local-repair versus rebuild policies under tiered storage, cross-shard freshness contracts for distributed vector stores, and stronger operational metrics that connect refresh, rebuild, and compaction decisions to end-to-end service SLOs rather than isolated retrieval scores.

4.2.5 Continual Learning and Retention Governance. Continual-learning systems are often read mainly through forgetting curves, but for this survey their more durable contribution is a runtime language for bounded retention. Once memory is scarce, the central question is no longer simply how to adapt online, but which historical state deserves protection, replay, compression, or eviction as future updates continue to arrive. Constraint-based methods such as EWC treat previously learned parameters as protected state, while GEM and A-GEM reinterpret an episodic buffer plus gradient projection as a runtime control over update interference [27, 111, 145]. Exemplar-based methods such as iCaRL expose a complementary retention surface: once a bounded exemplar set also anchors class statistics and distillation targets, admission policy is no longer only about remembering old samples, but also about stabilizing the representation space that future updates will inherit [182]. Replay-centric methods shift the control surface again: experience replay, MIR, DER, and GDumb all assume that bounded historical memory is inevitable, then ask which samples should be admitted, replayed, distilled, or retained to maximize future utility rather than merely preserve recency [9, 15, 175, 185]. REMIND adds a representation-level variant by showing that feature-space replay and lightweight reconstruction can make retention cheaper without eliminating the underlying state-budget problem [75]. As a group, these papers formalize retention as a governed state problem in which the runtime must decide what historical signal is worth protecting, how it is stored, and when it should influence future updates.

The discussion uses continual-learning papers selectively. It does not attempt to re-survey the whole continual-learning algorithm landscape, which is already well covered elsewhere [38]. Instead, it extracts the subset of methods that make bounded memory, replay eligibility, or retention budgets legible as runtime-governed state objects. That methodological narrowing keeps the discussion tied to runtime-governed retention surfaces rather than expanding into a general review of learning dynamics.

Read through a runtime lens, these methods already expose several non-interchangeable retention objects. EWC protects parameter-importance state and therefore treats retention as constrained parameter movement without materializing a replay store at all [111]. iCaRL instead turns a bounded exemplar set into a published memory object whose admission policy must stabilize both class prototypes and future representation quality [182]. GEM and A-GEM make episodic samples governable through replay-time interference constraints, while MIR goes one step further by ranking retained examples according to expected future gradient conflict rather than retaining them only because they arrived recently [9, 27, 145]. GDumb is useful precisely because it removes most of that controller complexity: once the method is reduced to admission-only sample curation, the field can see more clearly when final retained quality is dominated by the store's intake

policy instead of by sophisticated online update logic [175]. REMIND changes the same boundary by compressing replay into feature-space state, which means the effective retention budget is no longer just sample count but also representation-fidelity loss, refresh cost, and compaction debt [75]. The comparison is therefore not merely algorithmic. These papers govern different state objects and expose different control seams over what historical signal remains available to future updates.

Recent online systems make that operational reading harder to ignore because they separate responsibilities that older fixed-budget comparisons often blur together. One responsibility is surviving budget elasticity under ongoing writes and asynchronous updates, where controllers must tolerate abrupt contraction, pipeline interference, and changing write pressure rather than one static buffer size [281]. Another is ranking retained state by future utility instead of by recency alone, so admission decisions explicitly preserve downstream representativeness and reuse value [121]. Set beside MIR, DER, and GDumb, these systems show that continual retention is not only a forgetting problem. It is a bounded-store governance problem in which admission quality, replay cost, disturbance response, and budget elasticity jointly determine whether historical state remains worth keeping over long horizons [9, 15, 121, 175, 281].

A related line studies reusable training or federated state rather than inference-time memory. Obscura, Quiver, and SuperNeurons are closest to the training-runtime side of that question: previously materialized blocks, dataset shards, and activation tensors only remain valuable when admission, compression, and tiering policies preserve future training efficiency under memory pressure [86, 113, 212]. FLStore and AssyLLM sit at a different service boundary, where reusable objects are federated model blocks and assembly metadata whose compatibility and movement costs determine adaptation quality across participants [106, 247]. What links these lines is narrower but still important: reuse value is not inherent in the object; it must be surfaced to the runtime through ownership metadata, block compatibility, or tier-aware access paths. This broadens the evolution axis beyond continual-learning buffers alone and makes explicit that long-horizon reuse is equally a systems problem in large-scale training and federated adaptation.

The more reusable comparison is therefore not between named replay heuristics, but between four control surfaces with different evaluation boundaries. Protection methods ask which already-learned signal is expensive enough to freeze or regularize [111]. Admission methods ask which bounded samples or exemplars deserve to occupy scarce buffer space because they preserve future class coverage or reuse value [121, 175, 182]. Replay methods ask when buffered state should be invoked to absorb gradient interference and when its ranking cost outweighs its benefit [9, 27, 145]. Compression-oriented methods ask whether a cheaper representation can preserve enough future utility without turning refresh and reconstruction into the next bottleneck [75]. Ferret then makes budget elasticity itself part of the control plane by showing that even a good admission or replay policy can collapse when memory budgets shrink online and update pipelines keep writing [281]. The common systems lesson is that stable adaptation depends on explicit interference control and budget-aware lifecycle policy rather than on unconstrained online updating. The remaining gap is a retention-store contract that composes admission, replay, compression, demotion, and budget elasticity once these controllers begin to share infrastructure with retrieval caches, training checkpoints, or other long-lived state pools.

Read together, these papers already imply a retention lifecycle that the prose should make explicit. First, the runtime protects or publishes a subset of historical signal that must not be overwritten casually, whether that object is parameter importance state or a curated exemplar memory [111, 182]. Second, it decides which new evidence is admitted into the retained set under a bounded-store budget, as emphasized by GDumb and StreamFP [121, 175]. Third, it decides when retained state should be replayed, distilled, or otherwise reintroduced onto the update path, as in

GEM, A-GEM, MIR, and DER [9, 15, 27, 145]. Fourth, it decides whether older retained signal should be compressed or demoted before outright eviction, which REMIND makes explicit at the representation layer [75]. Finally, Ferret shows that the store budget itself may contract online, turning what looked like a local replay decision into a disturbance-time governance decision over what must remain protected, what may degrade, and what can be shed safely [281]. The missing systems piece is therefore not another isolated selector, but a retention-store contract with explicit precedence over protect, budget-shrink enforcement, compress/demote, admit, and replay actions so that retention quality, maintenance debt, and disturbance response are evaluated on the same service boundary. Under disturbance, the precedence should be asymmetric rather than egalitarian: the runtime should preserve protected signal first, enforce the contracted budget second, use compression or demotion before outright loss, defer new admission before sacrificing already-vetted state, and replay only after the store has re-entered a safe operating region. Without that ordering, one controller can keep injecting replay or new admissions while another tries to shrink or compact the same store, turning retention quality into oscillatory maintenance debt instead of a governed long-horizon asset.

Recent literature increasingly points to a retention-store governance problem rather than a narrow replay-design problem. EWC treats the protected set as a scarce budgeted object, GEM and A-GEM make replay eligibility a runtime choice under interference constraints, MIR and GDumb show that admission policy can dominate eventual retained quality, DER and REMIND turn stored representation itself into a maintenance surface rather than a free byproduct, and Ferret plus StreamFP make budget fluctuation and selection telemetry part of the runtime loop rather than background assumptions [9, 15, 27, 75, 111, 121, 145, 175, 281].

The broader reading consequence is that continual retention should be compared through governance questions analogous to those used for retrieval indexes or serving caches, not through identical accuracy metrics alone. Beyond task outcomes, the runtime must also budget storage, rank future utility, compact stale state, and account for maintenance debt over long-lived service horizons.

Open gaps include system-level integration: many continual-learning methods remain model-centric and do not specify operational semantics for distributed retention stores, update fault handling, or cross-service consistency. A second gap is lifecycle accounting: the literature is strong on forgetting metrics but much weaker on maintenance overhead, refresh debt, and retention-store behavior under long-running multi-tenant service conditions. A third gap is elasticity semantics: even recent online methods say relatively little about how retention policy should react to abrupt budget contraction, tier migration, or background compaction under shared-service load. Survey work in continual learning already makes clear that evaluation choice strongly shapes what retention behavior becomes visible at all [38]; from a systems angle, that means retention governance needs not only better mechanisms but also benchmark protocols that expose storage pressure, maintenance cost, and disturbance response rather than accuracy curves alone.

Figure 5 summarizes why these later sections belong together. Serving memory, retrieval memory, and retention stores have different lifetimes, but they all revolve around the same systems question: how a runtime uses shared telemetry and service boundaries to decide which state to admit, refresh, compact, or preserve. The figure therefore supports a stronger comparison than simple co-location in one section: it shows that the main difference across these memory planes is not whether control exists, but how quickly state value decays and how expensive it is to reverse a mistaken lifecycle decision.

That is also why the manuscript pauses for matrices at this point. The prose has already argued that the same control logic recurs across domains; the tables below make that claim auditable at

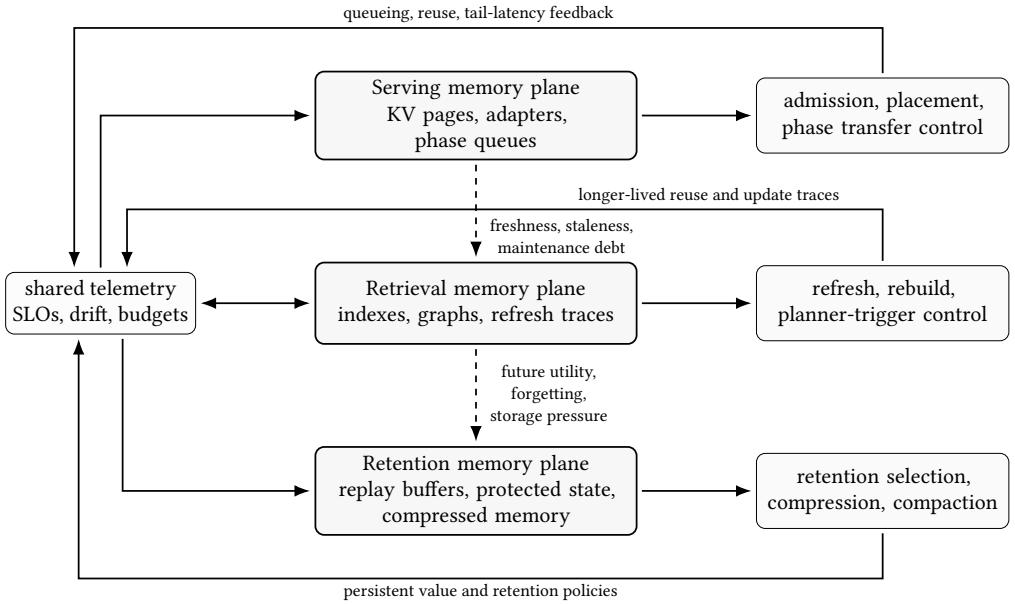

Fig. 5. Memory governance planes across serving, retrieval, and continual retention. The state objects differ in lifetime and granularity, but the control loop is structurally similar: shared telemetry feeds admission or rebuild decisions, and those decisions feed back into future latency, freshness, and reuse value.

larger scale by compressing mechanism families, boundaries, and blind spots into one scanable reference layer.

4.2.6 Comparative Mechanism Matrices. The following long-form matrices make cross-domain comparison explicit at scale and complement the narrative synthesis with structured side-by-side evidence. They are meant as reference scaffolds rather than stand-alone argumentative units: the main claims of the survey remain in the prose comparisons above and below, while the matrices compress recurring mechanism families, evaluation boundaries, and open gaps into a form that can be checked quickly across subfields.

Table 5. Large-scale mechanism matrix across representative papers.

Paper line	Dominant state object	Primary control surface	Evaluated boundary	Extension note
Aurora/Borealis [1, 2]	continuous-query operator state	graph adaptation control	streaming throughput/latency	compare with modern migration control
MillWheel/Naiad/ Dataflow [7, 8, 157]	keyed window and progress state	time/progress semantics	correctness-latency tradeoff	add reconfiguration stress
Trill/Differential/ Flink/StreamCloud [20, 23, 69, 153]	incremental state and progress metadata	frontier progress and elastic ownership	update latency and operability	connect progress with migration/recovery
BriskStream/ Concurrent stream [257, 262]	shared access paths and sync state	topology-aware placement and partitioning	throughput under contention	add multi-tenant hot-key stress
MorphStream [151]	transactional dependency state	dependency-aware scheduling	latency/throughput stability	compare with static baselines
Fast recovery line [270]	recovery metadata and logs	recovery-integrated control	recovery time and steady-state overhead	unify with migration control

Paper line	Dominant state object	Primary control surface	Evaluated boundary	Extension note
Megaphone/ OSM/SEEP [21, 51, 160]	movable state shards	migration and elasticity control	reconfiguration latency	add correctness-envelope taxonomy
FineStream/CStream/ CompressStreamDB [245, 249, 265]	representation-aware operator and format state	hardware-conscious mapping + state-path control	speed/energy-delay tradeoffs	extend to accelerator mixes and closed-loop format switching
PECJ/LibAMM [243, 246]	quality-sensitive intermediate state	approximation and compensation control	bounded quality vs cost	unify with service-level quality floors
vLLM [115]	paged KV and memory-plane state	allocation, sharing, and lifecycle admission	throughput at latency target	add cluster lifecycle governance
Sarathi-Serve [6]	phase-skewed serving state	prefill/decode scheduling and batch shaping	throughput-latency frontier	benchmark disturbance and phase migration
SGLang [275]	reusable program-structured state	radix reuse and program-boundary control	throughput on structured programs	map mixed-shape cluster governance
Clockwork/ Clipper [35, 68]	serving control-plane state	predictability and orchestration control	tail-latency/SLO isolation	tie to lifecycle admission
DistServe/Splitwise/ FastGen [79, 172, 278]	disaggregated phase and transfer-queue state	phase split and cluster governance	goodput + tail behavior	unify with tenant-aware admission
Nexus/InferLine/ INFaaS/Punica/ AlpaServe [30, 34, 126, 186, 192]	multiplexed tenant and control-plane state	admission, provisioning, isolation, and placement control	throughput/cost/ SLO stability	unify phase and tenant governance
DPR/ColBERT [105, 107]	dense and late-interaction retrieval indexes	offline/online retrieval decomposition	retrieval quality vs serving cost	add update/refresh lifecycle dimension
Self- RAG/FlowRAG/ CANDOR [12, 213, 259]	planner and evolving index state	retrieval gating + rebuild control	long-horizon quality/ cost balance	add staleness-trigger policy
Milvus/Manu/ FreshDiskANN/ IP-DiskANN [71, 196, 211, 228]	segmented vector-store and mutable graph state	shard lifecycle, deletion, and maintenance control	distributed latency/ freshness balance	define cross-shard freshness and compaction contracts
HippoRAG/ FlowRAG [218, 259]	graph-enhanced and evolving retriever memory	dynamic memory evolution control	long-horizon quality and efficiency	add stale-memory debt metrics
Continual retention line				

Paper line	Dominant state object	Primary control surface	Evaluated boundary	Extension note
[9, 15, 75, 111, 121, 145, 175, 281] Approx. analytics line	constrained parameters, replay buffers, compressed state, and budget traces	interference control, replay selection, budget elasticity, and retention governance	forgetting vs adaptation quality + store cost	connect with retrieval retention, maintenance debt, and elastic shared stores
[5, 31, 64, 82, 214] Spanner/Calvin/ FaRM line	summary, sample, and compensation state	accuracy-cost budget and maintenance control	latency/error boundaries	link with serving SLAs and recovery-time error contracts
[33, 42, 204] CRDT and replicated-data line	distributed transaction state and replication metadata	consistency/placement/ concurrency control	correctness + latency + scale	compare with serving memory ownership models
[191]	convergent replicated object state	conflict-free merge semantics	convergence and availability	study with retrieval-memory updates

Table 6. Evaluation-and-gap matrix across representative state-management clusters.

Cluster	Recommended disturbance tests	Required metrics	Common blind spots	Open research direction
State access under skew	hotspot flip, burst tenant switch, rebalance events	throughput, p99, conflict density, migration lag	stationary-load-only evaluation	add workload phase annotations to results
Recovery-integrated scheduling	fail-stop at different load phases, replay stress	recovery time, post-recovery stability, backlog clearance	treating recovery as isolated mode	quantify steady/recovery coupling cost
Elastic stream migration	scale-out/in under active joins/windows	migration overhead, latency amplification, ownership transfer delay	no ownership semantics during transfer	formalize ownership-transfer legality under disturbance
Hardware-conscious execution	topology perturbation and device contention	energy per useful result, latency, quality floor	microbenchmark-centric claims	add end-to-end boundary plots
Approximation control	drift + disorder injection	bounded error, utility loss, throughput	static error assumptions	evaluate adaptive error budgeting
KV cache lifecycle	long prompts + bursty decode + mixed sampling modes	memory waste, reuse ratio, p99, admission drop rate	allocator-only optimization view	add lifecycle-stage breakdown figures
Serving disaggregation	independent pressure on prefill/decode pools	goodput, queue imbalance, transfer overhead	homogeneous-request assumptions	build mixed-length benchmark suite
Structured serving programs	control-flow-heavy prompt programs	reuse hit rate, decode constraint overhead, tail latency	ignoring program-shape diversity	add program-template taxonomy

Cluster	Recommended disturbance tests	Required metrics	Common blind spots	Open research direction
Retrieval index evolution	continuous corpus updates + stale data injection + ANN churn + deletion bursts	recall@k, answer quality, update cost, query latency, freshness lag, compaction debt, cross-shard skew	one-shot index build assumption	define refresh-compaction-rebuild schedules with shard exposure rules
Graph memory retrieval	multi-hop drift and conflicting evidence injection	quality robustness, latency, maintenance overhead	no maintenance debt accounting	add graph maintenance cost analysis
Continual retention policies	memory budget sweep with domain shift + abrupt budget contraction	forgetting, adaptation speed, retention overhead, compaction lag, demotion loss	model-only reporting without system costs	add retention-store systems metrics and elasticity protocols
Cross-domain middleware	concurrent serving + retrieval + update workloads	SLA violations, cross-layer backpressure, policy conflicts	layer-by-layer optimization	compare policy-composition interfaces under mixed workloads
Tiered memory architectures	hot/cold transition storms	promotion accuracy, movement cost, hit/miss latency gap	simplistic hotness heuristics	evaluate prediction-based tiering policies
Quality-bounded serving	quality floor shifts at runtime	quality, latency, throughput, fallback frequency	fixed-quality-threshold assumptions	add dynamic bound adaptation study
Scheduling-policy composition	overlapping policy triggers	oscillation frequency, convergence time, rollback count	no conflict semantics	define policy precedence model
Observability fidelity	delayed/noisy metrics simulation	decision regret, control delay, wrong-action rate	perfect telemetry assumption	add signal-confidence modeling subsection
Admission governance	adversarial burst and tenant unfairness workloads	admission fairness, SLA stability, rejection utility loss	average-only admission metrics	include per-tenant fairness analysis
Long-horizon reuse quality	week-scale synthetic drift phases	quality decay slope, refresh debt, maintenance spikes	short-run benchmark horizon	add multi-phase evaluation protocol

Viewed side by side, the two matrices surface three bounded comparison results rather than one grand synthesis. First, comparable control seams recur even when the state object changes: visibility and ownership dominate streaming, lifecycle and transfer dominate serving, and maintenance debt dominates retrieval and retention. Second, evaluation blind spots are remarkably stable across domains, especially around disturbance phases, delayed telemetry, and deferred maintenance cost. Third, the most reusable open problems remain contract problems rather than domain-specific features: how to define exposure safety, policy precedence, disturbance-aware accounting, and transfer semantics in a way that survives movement across layers. This comparison should not be read as claiming identical service boundaries across domains; it claims instead that these literatures repeatedly expose structurally similar governance questions once each row is reduced to the same comparison unit of state object, control surface, evaluated boundary, and remaining seam.

The tables therefore serve as compressed comparison aids for the surrounding synthesis, not as replacements for it.

4.3 Analytical Maturity

4.3.1 Cross-Layer Mechanism Synthesis. Across access, execution, and evolution, the literature reveals a recurring systems pattern: local policy gains often disappear once effects propagate through the state lifecycle. In streaming systems, fine-grained synchronization and partitioning can improve multicore scalability [256, 262], yet these gains degrade when topology and locality are not co-optimized [257]. Hardware-conscious lines similarly show that movement and representation of state can dominate kernel-level arithmetic gains [245, 249].

Inference and workflow runtimes exhibit the same mechanism once state becomes visible enough to steer execution. Paged KV management and reuse-aware serving turn memory from an implementation detail into an explicit control object [115, 275], yet allocator improvements alone are insufficient when phase asymmetry between prefill and decode remains unscheduled or when reuse legality remains implicit under transfer and restoration [6, 79]; this mirrors transactional stream findings where stable behavior requires dependency-aware online control rather than static plans [151]. A nearby shift is from request-centric control to workflow-aware control. SONIC and ORION showed in serverless DAGs that once runtimes expose dependency-local data passing, stage bundling, and warm-start state, the dominant optimization target becomes end-to-end workflow completion rather than isolated invocation latency [147, 148]. Parrot and SGLang reach a comparable conclusion inside LLM systems from two different interfaces: Parrot lifts semantic variables, request DAGs, and application objectives into the serving control loop, while SGLang turns reusable prefix structure and constrained decoding into program-visible runtime state [132, 275]. What is shared is narrower than a common serving mechanism: performance improves only after the runtime can see intermediate dependency state and schedule around it explicitly, not when it treats each call as an unrelated unit.

Recent multi-tenant serving, approximation, retrieval, and recovery lines sharpen the same cross-layer lesson from different directions. S-LoRA, Jenga, and Prism show that once adapters, heterogeneous cache layouts, and whole-model residency share one GPU pool, the real control problem is no longer per-request KV placement alone, but coordinated ownership over adapter pages, allocator state, and model activation windows [193, 236, 248]. Approximation and retrieval lines reinforce the same boundary-first logic: approximate kernels and compensation mechanisms are sustainable only when quality floors are integrated into runtime decisions [243, 246], while long-horizon retrieval and retention systems show that update and maintenance policies directly shape downstream quality-cost trajectories [259, 281]. Recovery and elasticity complete the picture because snapshotting, migration, and recovery acceleration are most robust when they share transfer semantics and observability channels instead of being isolated subsystems [19, 51, 270]. The recurring principle that survives these comparisons is therefore state-as-scheduling-object: many modern runtimes schedule around mutable state objects rather than tasks over passive data, and explicit lifecycle stages (admit, place, mutate, expose, compact, reclaim) make that governance problem comparable across otherwise different domains [132, 147, 148, 172, 275, 278].

Persistent unresolved mechanism gaps include:

(1) high-fidelity observability for semantic state conflicts beyond coarse counters;
(2) cross-tier lifecycle coordination for short-lived and long-lived memory objects;
(3) policy-composition semantics that prevent oscillation under overlapping triggers;
(4) long-horizon accounting of maintenance debt and reuse-value decay.

These four gaps cluster around one underlying problem: current runtimes still expose too many local mechanisms without exposing the inter-layer contracts needed to make those mechanisms compose over time.

These unresolved gaps also provide a sharper evidence hierarchy for the literature itself. In the rubric used by this survey, a paper contributes reusable systems evidence when it does more than report a local speedup: it identifies the governed state object, exposes a control surface over that object, shows how the resulting policy survives disturbance or composition pressure, and makes the optimized service boundary explicit. Mechanisms that succeed on only the first one or two steps still matter, but they should be read as partial closures of a control seam rather than as evidence that the seam is solved. Under that standard, transferability is a qualified judgment about mechanism structure and evaluation boundary, not a claim that results automatically carry over unchanged from one workload family to another.

4.3.2 Evaluation Protocols and Threats to Validity. Evaluation should treat state management as closed-loop control, not isolated optimization. Each experiment should explicitly specify the state object, control surface, and service boundary before reporting outcomes. Otherwise, local speedups may be mistaken for end-to-end gains even when lifecycle paths remain unchanged [6, 115, 151, 275].

Disturbance-driven evaluation should be treated as a baseline requirement because stationary replay alone hides the instabilities that appear under burst, skew, and phase transitions. Protocols should fix random seeds, preserve disturbance timestamps, and report both phase-local and aggregate outcomes so inflection points are auditable [257, 262]. In the surveyed settings, this is the most reliable way to separate a controller that truly absorbs disturbance from one that merely defers visible debt into a later phase of the run. A minimal protocol suite should include:

(1) **burst disturbance**: short high-intensity arrivals stressing fragmentation and queue growth;
(2) **skew disturbance**: popularity flips testing migration and reuse stability;
(3) **topology disturbance**: controlled device/NUMA perturbations isolating movement sensitivity;
(4) **semantic disturbance**: update drift probing retention and index-evolution coupling.

Long-horizon metrics are necessary when claims involve reuse or adaptation. Examples include reuse survival, reclamation debt, update-to-benefit lag, post-shock recovery half-life, and quality-retention trajectories; these reveal delayed regressions that short-window throughput cannot capture [115, 259, 281]. For retrieval and retention systems, the protocol should be made even more concrete: retrieval evaluations should distinguish local shard freshness from cross-shard exposure freshness, and retention evaluations should distinguish nominal retention budget from effective retention budget after background compaction, demotion, or migration. Coupling-aware ablations should also be standard, because many gains emerge only when access, execution, and evolution controls are coordinated [151, 245, 246]; the same experiments often reveal the opposite case, where an optimization that looks strong in isolation vanishes once neighboring controllers consume the same telemetry channel or disturbance-response window. Finally, negative results should be first-class evidence. Reports should include boundary failures, sign reversals (for example, higher reuse but worse p99), and oscillation windows under disturbance so that claim scope is tightened to measured envelopes rather than to the most favorable regime [224, 246, 262].

Threats to validity should be operationalized as constraints: internal validity via seed/time control and warmup policy; construct validity via strict separation of overlap opportunity vs realized reuse; external validity via multi-regime and topology-perturbed replication. Claims should remain bounded to tested disturbance and lifecycle horizons.

Table 7. Reader guide for translating the synthesis into runtime design choices.

Recurring synthesis result	Design question	Concrete artifact to look for
Observability gaps dominate failure analysis	Which signal names the state object, lifecycle stage, delay semantics, and legal action?	typed telemetry and disturbance indicators
Local speedups disappear under composition	Which boundary is hard, which objective is soft, and who wins on conflict?	precedence-aware decision kernel
Lifecycle debt accumulates across updates and movement	Where are admit, place, mutate, compact, expose, and evict made explicit?	lifecycle pipeline with bounded actuation interfaces
Recovery reuses the same state objects as steady state	Which invariants survive migration, replay, rebuild, or restore?	safety envelope and ownership-transfer rules

4.3.3 Research Agenda Beyond Isolated Policy Tuning. The next stage of progress is architectural rather than heuristic: isolated policy tuning can improve local regimes, but cross-layer instability persists when lifecycle governance is fragmented [6, 115, 151, 257]. Across the literature, the recurring requirements are already visible: observability that links local contention to service drift, scheduling that treats mutable state rather than tasks as the primary placement object, recovery-aware lifecycle control, and explicit semantics for composing overlapping policies. The retrieval and retention lines make this especially concrete: future systems need benchmarkable contracts for when a partially refreshed shard may serve traffic, when a retention tier may demote memory under pressure, and how deferred maintenance debt is surfaced before it causes visible quality collapse.

These requirements motivate the next two sections. Section 5 turns them into design implications and runtime contracts; Section 6 narrows them into evaluation norms and a shorter forward-looking agenda. The main point here is comparative rather than prescriptive: the same unresolved control seams recur across streaming, serving, retrieval, and retention, which suggests that the field is ready to converge on shared runtime abstractions rather than continue proliferating domain-local heuristics.

5 DESIGN IMPLICATIONS FOR STATEFUL RUNTIMES

The comparative synthesis now supports a bounded set of design implications. Rather than proposing one settled architecture, this section extracts the recurring principles, design axes, anti-patterns, and reference abstractions that appear necessary when stateful mechanisms must remain composable under disturbance.

Table 7 provides the intended reading order for this section. The argument now moves from principles, to architectural axes, to anti-patterns, and finally to a minimally complete reference blueprint, with each step narrowing the gap between comparative survey judgment and concrete runtime design without claiming that one canonical architecture has already emerged.

5.1 Cross-Cutting Design Principles

The literature reviewed above points to several reusable principles. They are best read not as slogans but as recurring design constraints that successful stateful systems rediscover under different workloads and hardware settings. Each principle is also a direct response to one or more

anti-patterns from the earlier sections, which is why they are phrased as operational commitments rather than abstract architectural preferences.

5.1.1 Observation Before Optimization. Across the surveyed domains, optimization becomes more credible once the system makes the relevant state boundary observable. In streaming and transactional dataflow, the decisive shift was exposing queues, synchronization paths, progress metadata, and dependency structure instead of treating contention as unexplained throughput loss [151, 256, 262]. In approximation-aware execution, the analogous requirement is to expose error floors, compensation state, and quality debt as runtime-visible signals rather than offline benchmarking artifacts [243, 246]. Retrieval and retention systems surface the same principle under different names: freshness lag, rebuild debt, replay eligibility, and effective memory budget generally need to be visible before a controller can decide whether to refresh, compact, demote, or preserve state [213, 259, 281]. The common lesson is that observability is most useful when signals already encode a legal control surface.

This is why observation should be treated as a design primitive rather than a later operational layer. More robust systems expose telemetry that is typed by lifecycle stage, delay semantics, and actionability. Once framed this way, a frontier in streaming, a phase-imbalance signal in serving, and a freshness indicator in retrieval become comparable architectural objects because all three determine whether the runtime can make a boundary-safe decision. Visibility is therefore not just the precondition for control; it is the first contract that makes control interpretable.

5.1.2 State as a First-Class Scheduling Object. Traditional schedulers place tasks on resources and treat data as passive input. The literature reviewed here points in the opposite direction: modern runtimes increasingly schedule around mutable state objects whose location, shape, and future reuse value dominate performance. Stream engines already made this visible through key partitions, operator shards, and dependency state that determine whether concurrency helps or hurts [151, 257]. LLM serving systems push the same principle into memory-bound deployment, where KV pages, adapter state, partition metadata, prefill/decode working sets, and model-residency slots are the objects that admission, batching, and placement policies are really coordinating around [6, 30, 115, 126]. Recent multi-model serving systems sharpen the same point: S-LoRA, Jenga, and Prism show that once adapters, heterogeneous cache layouts, and whole-model activation windows contend for the same GPU pool, scheduling must reason over residency and ownership directly rather than treat memory pressure as a secondary byproduct of request placement [193, 236, 248]. Retrieval and retention systems generalize the idea further because vector shards, graph memory, replay buffers, and summary state remain useful only when the runtime can decide which state deserves fast access, protected residency, or delayed maintenance [121, 218, 259].

The deeper implication is that scheduling should be formulated over state-object lifecycles rather than over isolated requests. Once this shift is made, batching becomes a question of which state can be reused safely, migration becomes a question of when moving state is worth future interference reduction, and placement becomes a question of who should inherit ownership of a hot or fragile state object. This state-centric view is the clearest common abstraction linking classic stream runtimes with newer memory-heavy AI services.

5.1.3 Closed-Loop Rather Than Open-Loop Policies. Across access management, execution control, and state evolution, more robust systems differ from weaker ones less by whether they use heuristics or models than by whether they close the loop between observation, decision, action, and post-action accounting. MorphStream adapts scheduling against observed dependency shape rather than assuming one execution plan remains valid under all transaction mixes [151]. PECJ

closes the loop by feeding disorder and quality loss back into compensation decisions instead of treating bounded error as a one-time offline guarantee [246]. Serving systems such as Sarathi-Serve, DistServe, and CacheFlow make the same point at cluster scale: admission, phase splitting, transfer, and restoration cannot remain independently tuned because each changes the future memory pressure that subsequent policies inherit [6, 162, 278]. Retention-oriented systems such as Ferret and StreamFP similarly show that admission and replay policies remain brittle when budget changes and future-utility signals are treated as static assumptions [121, 281].

The practical consequence is that open-loop policies should now be read as local approximations, not as complete runtime designs. A sampling rule, compression level, or eviction heuristic may work well inside one disturbance regime, yet fail once another controller changes the workload shape or ownership pattern that the heuristic implicitly assumed. Closed-loop design is therefore not merely an aesthetic preference; in long-lived, shared, and continually perturbed settings, it is often the most reliable way to keep policy interactions auditable.

5.1.4 Stable Gains Require Boundary Modeling. Several prominent papers in the survey report mechanisms that are genuinely effective but only within an explicitly modeled service boundary. Clockwork, vLLM, and related serving systems matter not merely because they improve latency or throughput, but because they clarify which latency targets, batching regions, and memory assumptions define the regime in which those gains hold [68, 115]. Approximation-aware systems make the same lesson more explicit: a method is only meaningful if it states which quality floor, error envelope, or downstream sensitivity remains protected while cost is reduced [243, 246]. Retrieval and retention systems add a longer-horizon variant of the same issue, where refresh cadence, compaction policy, or budget demotion appear beneficial until the unmodeled cost is paid later as rebuild debt, quality drift, or degraded future reuse [213, 259, 281].

Boundary modeling is therefore the main separator between local optimization and infrastructural contribution. A result that improves one metric without declaring what is held fixed, what debt is deferred, and what disturbance invalidates the claim is difficult to compose with other mechanisms. By contrast, a well-bounded mechanism can be integrated, stress-tested, and compared across domains. Efficient state management depends at least as much on identifying safe operating regions and failure envelopes as on optimizing within those regions.

Taken together, these principles say less about one ideal scheduler or storage layer than about the interfaces that repeatedly appear when local mechanisms remain legible under composition. The next subsection therefore shifts from normative principles to architectural choices that the literature keeps rediscovering: where control lives, at what granularity state is represented, and how lifecycle stages become explicit runtime objects.

5.2 A Design Space for Stateful Runtime Architectures

The previous sections established recurring control seams; this section recasts them as a practical architecture design space. It makes explicit which design decisions tend to co-travel in successful systems and which combinations repeatedly cause fragility. We focus on four architectural axes: state granularity, control-plane coupling, lifecycle management, and service-boundary alignment.

5.2.1 State Granularity: Record, Key, Segment, and Session. A first architectural choice is the granularity at which state is represented and controlled. Record-level granularity offers maximal flexibility but often suffers from high metadata overhead and weak locality. Key-level granularity, common in stream processing, can align naturally with partitioning and migration but can become skew-prone under hot keys [150, 257]. Segment- or page-level granularity, prominent in modern LLM serving, trades finer control for stronger packing and transfer efficiency in memory-bound paths [6, 115]. Session- or conversation-level granularity, common in agentic and retrieval-heavy

systems, simplifies policy interfaces but can hide substantial internal heterogeneity in reuse value and retention cost [218, 275].

The most robust systems usually support more than one granularity and explicitly model conversion cost between them. For instance, a runtime may collect telemetry at key-level but schedule migration at chunk-level, or retain retrieval memory at document-level while serving lookups at token-level interaction granularity [107]. A recurring failure mode is locking the entire system into one granularity because an early implementation made it convenient. That shortcut often appears efficient at prototype scale, then becomes a hard scalability ceiling once workloads diversify.

5.2.2 Control-Plane Coupling: Embedded, Sidecar, or Shared Service. A second choice is where state control logic lives. Embedded control keeps policy decisions near execution and can minimize coordination latency. However, it often creates fragmented policy semantics across subsystems. Sidecar-style control improves modularity and operability but can introduce lag between telemetry and action if the sidecar does not receive sufficiently rich state-change signals. Shared memory-control services offer stronger global consistency and reuse across workloads, yet risk becoming bottlenecks unless their APIs are designed around coarse enough operations and composable intents.

Historical systems illustrate each tradeoff. Stream engines with deeply embedded scheduling and state-placement logic can react quickly to contention but often need substantial refactoring when recovery and migration policies evolve [262, 270]. Serving systems that elevate memory control (for example, KV paging or phase-aware admission) to clearer runtime layers gain better policy visibility but must invest more in interface stability and backpressure semantics [6, 79, 115]. Retrieval-memory stacks frequently begin with loosely coupled indexing services, then move toward tighter control-plane integration once update cadence and quality drift become dominant concerns [105, 259].

5.2.3 Lifecycle Management: Admit, Place, Mutate, Compact, Evict. Stateful runtime design is often framed around placement only. In practice, robust systems require an explicit lifecycle pipeline with at least five stages: admission, placement, mutation, compaction, and eviction. Admission controls whether new state should enter the active working set. Placement decides where it resides relative to compute paths. Mutation governs update semantics and write amplification. Compaction controls structural debt accumulated by updates and deletions. Eviction determines which state leaves the active tier and under what quality or recovery constraints.

This lifecycle framing unifies otherwise separate communities. In stream processing, admission can correspond to which keys or windows are promoted into fast paths; mutation corresponds to incremental state update and checkpoint integration; compaction appears in state backend maintenance and log trimming. In LLM serving, admission maps to request acceptance under memory pressure; placement maps to KV page residency; mutation maps to decode-time growth and prefix-share edits; compaction and eviction appear as memory reclamation and reuse-window tuning. In retrieval systems, admission maps to indexing policy under dynamic corpora; placement maps to tiered vector storage; mutation maps to embedding refresh and graph rewiring; compaction maps to index maintenance; eviction maps to stale-memory retirement.

The newer multi-model serving literature suggests that this mapping is still too coarse if it stops at per-request KV state. In S-LoRA, Jenga, and Prism, admission increasingly includes deciding whether an adapter, heterogeneous cache allocation, or even a whole model instance should remain active on the GPU at all; placement includes not only page residency but also which allocator region, virtual-memory reservation, or pooled engine instance owns the live state; and eviction becomes a service-level choice about when reclaiming memory is worth the future reactivation penalty [193, 236, 248]. This makes serving memory look less like a cache replacement problem

and more like a short-horizon residency management problem with explicit ownership transfer and SLO debt.

When any one stage is implicit, systems become difficult to reason about under stress. For example, if compaction policy is absent from the design model, mutation cost can silently dominate over time and negate throughput gains that were visible in short benchmarks. Similarly, if admission is treated as an outer load-shedding mechanism rather than a first-class state policy, latency can remain unstable even after aggressive scheduling optimizations.

5.2.4 Service-Boundary Alignment: Local Wins vs End-to-End Wins. A final architectural axis is alignment between local optimization targets and user-facing service boundaries. Many papers report local speedups that are real but operationally fragile because they optimize inside an unmodeled boundary. Boundary-aware systems are explicit about what is held fixed (latency percentile, quality floor, recovery objective) and what is allowed to move (throughput, energy, memory footprint). This distinction is central in approximation-aware systems [243, 246], memory-bound serving runtimes [68, 115], and dynamic retrieval stacks where quality drift accumulates over time [12, 218]. The architectural implication is clear: design reviews should require a written boundary contract for each state mechanism that specifies which service metric it claims to improve, under which workload assumptions, and with what fallback behavior when those assumptions fail. Absent this contract, integration teams tend to inherit brittle local optimizations with unclear operational envelopes.

The division of labor should now be clear. The design-space discussion identifies which choices a runtime must make explicit; the next subsection asks what predictably goes wrong when those choices remain implicit, are layered without precedence, or are evaluated under unrealistically narrow conditions.

5.3 Failure Modes and Anti-Patterns

Survey papers often focus on successful mechanisms and under-discuss recurring anti-patterns. An explicit anti-pattern taxonomy is useful because it transforms ”what to avoid” into a reusable review checklist. This matters for stateful systems in particular because many failures are not isolated bugs; they are repeated contract omissions that surface across streaming, serving, retrieval, and retention under different terminology.

5.3.1 Telemetry-Action Mismatch. A common failure is collecting state metrics that cannot trigger actionable runtime decisions. The literature increasingly shows that raw counters are insufficient once state-management decisions become boundary-sensitive. In stream systems, queue growth or lock contention matters only when it can trigger a concrete response such as repartitioning, migration, or a change in execution grain [256, 257]. In serving systems, cache occupancy or queue depth is informative only when the runtime can map it to admission control, phase splitting, or transfer scheduling [6, 278]. In retrieval systems, freshness indicators are useful only when they distinguish a locally healthy shard from one that is still unsafe to expose globally after rebuild or compaction [213, 259]. Telemetry that cannot be translated into a legal actuation target remains a dashboard artifact rather than a control input.

This anti-pattern appears whenever observability is designed independently from the policy layer. Systems that avoid it usually expose not only a metric value, but also its delay semantics, confidence, and the specific state object to which an action would apply. From a survey perspective, the key evaluative question is therefore not how many metrics a system exports, but whether the exported signals are typed strongly enough to support auditable runtime decisions.

5.3.2 Policy Layering Without Conflict Semantics. Another anti-pattern is policy accretion without conflict resolution. Teams add contention mitigation, then add migration heuristics, then add quality guards, each locally justified. Without explicit precedence or composition semantics, the resulting policy stack oscillates because each controller reasons from a different boundary. Stream systems illustrate this clearly: topology-aware placement, migration control, and recovery acceleration can all be locally sensible while still interfering if the runtime does not specify which objective dominates under skew or failure [160, 257, 270]. Serving systems expose the same issue under different names, where batching efficiency, admission control, and tenant isolation can pull in opposite directions unless latency objectives, memory limits, and fairness rules are ordered explicitly [30, 68, 278]. Retrieval-control systems are beginning to encounter the same problem as planner-mediated invocation, refresh scheduling, and shard-maintenance policies increasingly overlap [12, 259].

The literature suggests that stable systems do not avoid multiple policies; they make policy hierarchy explicit. Hard constraints, soft objectives, and tie-breakers generally need to be ordered so that a later controller cannot silently invalidate the assumptions of an earlier one. Without that conflict semantics, adding more intelligence to the control plane often increases instability rather than robustness.

5.3.3 One-Shot Benchmarking of Non-Stationary Mechanisms. Many state mechanisms are fundamentally temporal, yet are evaluated on short stationary runs. This mismatch is particularly harmful for retention and retrieval policies where degradation appears only after update cycles [259, 281]. The same danger also appears in recovery and disaggregation work: a scheduler that looks stable in a short replay may degrade sharply once burst, migration, and restoration debt are allowed to interact over time [162, 270]. Robust evaluation should therefore include warmup, disturbance, adaptation, and re-stabilization phases rather than a single averaged steady-state window.

This anti-pattern persists because one-shot experiments often validate a local mechanism while suppressing the lifecycle debt that mechanism creates. A survey-level reading should therefore treat short stationary wins as provisional unless the paper also reports phase-local failures, disturbance response, or long-horizon maintenance cost. Otherwise, improvements that disappear under drift or shock can look stronger than they are operationally.

5.3.4 Ignoring Write Amplification in "Read-Optimized" Designs. Read-side acceleration often hides expensive write-side debt. Retrieval literature makes this anti-pattern especially visible: PQ and HNSW can improve steady-state recall-latency tradeoffs, yet under continuous updates the real bottleneck often moves to graph repair, deletion handling, compaction, and rebuild cadence [93, 149, 196, 228, 229]. Production-oriented vector systems such as Milvus and Manu show that background indexing, segment sealing, and shard maintenance are not secondary plumbing; they define whether a read-optimized design remains operable under write churn [71, 211]. Once those costs are counted, some apparently efficient read paths turn out to be debt-shifting mechanisms whose gains depend on maintenance being pushed outside the reported window.

Serving systems reveal a shorter-lived analogue of the same issue. Paging, disaggregation, transfer compression, and restoration scheduling can improve TTFT or goodput, but they also create write-path or movement debt in the form of KV transfer, restoration pressure, and reclamation complexity [73, 115, 162, 172]. Stream and approximate systems show a comparable pattern when incremental summaries or compressed state paths reduce read-time work while silently increasing mutation and maintenance overhead [245, 246]. A mechanism should therefore not be considered efficient if it externalizes cost into deferred compaction, index rebuild, restoration debt, or retrain-

ing windows. The relevant comparison is whether the paper charges write amplification to the same service boundary as the reported read gains.

5.3.5 Treating State Ownership as Static. Ownership assumptions that are valid at small scale often fail under bursty multi-tenant workloads. Stream systems demonstrate this most clearly: once scale-out, migration, and replay are triggered by the same workload phase, static ownership of operator state becomes a liability because it forces elasticity and recovery to maintain separate transfer logic [51, 150, 160]. The stronger systems in this line increasingly separate logical ownership from the physical location of the state shard so that migration grain, replay path, and scheduling policy can evolve without redefining correctness from scratch.

Serving systems point toward the same conclusion from another direction. Phase-disaggregated serving, transferable KV state, and multi-tenant adapter management all imply that the runtime must decide who currently owns a short-lived state object, who may borrow or restore it, and when that ownership may move across workers or clusters [30, 172, 177, 278]. Retrieval systems make the issue visible again at shard granularity, where partially refreshed or compacted segments cannot be treated as statically owned if exposure policy must vary by freshness state. Designs that explicitly represent transferable ownership and bounded lease semantics therefore appear better positioned to adapt under dynamic pressure, although they require stronger correctness reasoning and clearer safety envelopes.

Across these anti-patterns, one design requirement emerges clearly: most failures stem from missing or implicit boundary contracts around observation, precedence, disturbance handling, write-path accounting, and ownership transfer. The blueprint in the next section should therefore be read as a direct response to these recurring failures rather than as an abstract implementation wish list. Its purpose is to force each control loop to answer five practical questions before deployment: what state is visible, what boundary is protected, what actions are legal, how conflicts are resolved, and which invariants still hold during disturbance.

5.4 A Contract-Oriented Blueprint for Stateful Runtimes

The blueprint below converts the synthesis into an implementation-oriented reference design. It is meant to summarize the minimum contract surface that recurrently appears across the surveyed mechanisms, not to prescribe one architecture or claim that current systems already implement a complete end-to-end stack in this exact form. Figure 6 therefore makes the blueprint explicit as a contract flow rather than a component checklist: the comparison point is whether a design exposes typed state objects, delayed telemetry semantics, precedence-aware decision logic, bounded actuation, and recovery-time guardrails clearly enough for neighboring mechanisms to compose.

5.4.1 State Catalog and Schema Contracts. A useful way to read the literature is that many of the stronger systems implicitly or explicitly approximate a machine-readable state catalog containing state object names, granularity, mutability class, lifecycle stage, and quality sensitivity. This catalog acts as a control-plane schema for runtime state. Without some equivalent abstraction, policy modules rely on implicit assumptions that quickly diverge across teams or subsystems. At minimum, the catalog should make ownership, exposure status, and allowed transitions explicit enough that an admission or migration policy cannot silently reinterpret what another controller believes the object to be.

For newer retrieval and retention systems, the catalog should also encode exposure and budget semantics explicitly. A retrieval shard should record not only whether it exists, but whether it is fresh enough to serve externally visible traffic after partial rebuild or compaction. A retention tier should record not only nominal retention budget, but effective retention budget after background

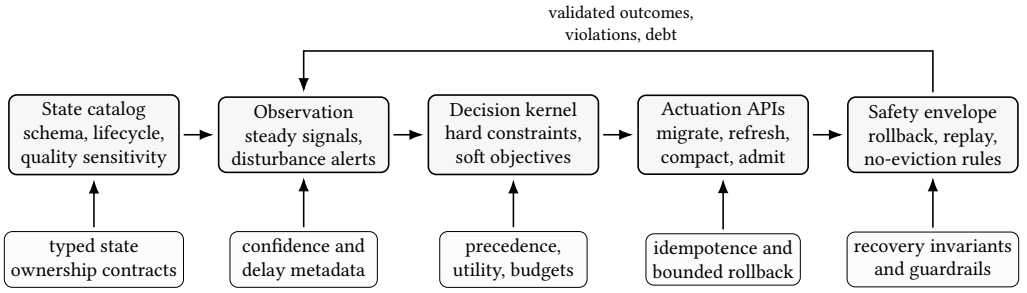

Fig. 6. Blueprint contract flow for stateful runtimes. The five layers are useful only when the contract between them is explicit: typed state feeds observation, observation feeds a boundary-aware decision kernel, decisions call bounded actuation interfaces, and a safety envelope validates outcomes and feeds violations back into the loop.

compaction, demotion, or replication overhead. These fields turn lifecycle debt into typed state rather than post-hoc debugging evidence.

The same catalog format can be used to map each reviewed system into a common state schema. Doing so exposes where papers are comparable and where they solve fundamentally different state problems. It also helps identify under-specified designs that report strong results without clear state typing. For survey writing, this schema is useful not only as an implementation suggestion but also as a discipline for deciding when two mechanisms belong in the same comparative paragraph.

5.4.2 Observation and Disturbance Detection. The surveyed systems suggest that useful observation layers include both steady metrics and disturbance indicators. Steady metrics capture normal load and quality behavior; disturbance indicators detect shifts such as hotspot flips, phase imbalance, retrieval drift, or memory-fragmentation spikes. This distinction becomes especially important when workload phases differ sharply, as in prefill/decode serving or periodic retriever refresh.

In practical memory-heavy services, this means distinguishing internal health from external exposure. Retrieval telemetry should separate local shard freshness from cross-shard exposure freshness, because partially rebuilt segments can be internally consistent yet unsafe to expose at the service boundary. Retention telemetry should likewise separate nominal retention budget from effective retention budget, because compaction lag, background migration, and replicated safety copies can silently shrink the space available for new samples.

An effective observation layer should expose confidence and delay metadata for each signal. Control policies that treat delayed telemetry as current truth are a major source of oscillation. Papers that make online control claims should therefore state these delay and confidence assumptions explicitly. In many systems, the decisive distinction is not whether a metric exists, but whether it arrives quickly and reliably enough to justify a legal action before the underlying state boundary has already moved.

5.4.3 Boundary Model and Decision Kernel. The decision kernel can be read as the point where state catalog and observation streams are turned into boundary-aware decisions. A useful abstraction is a constrained optimizer with two classes of inputs: hard constraints (SLA, quality floor, memory limit) and soft objectives (throughput, energy efficiency, migration overhead). The kernel can be heuristic, model-based, or learned, but the comparative requirement is that it produce auditable decisions and confidence estimates.

For systems papers, this is where many contributions differ: some improve signal quality, some improve model accuracy, some improve solver latency, and some improve safety under uncertainty. Presenting those differences through one decision-kernel abstraction makes cross-paper comparison much cleaner. It also clarifies whether two papers are solving the same control problem at different layers or whether they are optimizing against genuinely different constraint sets that should not be conflated.

5.4.4 Actuation Interfaces. Actuation interfaces are most comparable when they are explicit and idempotent where possible. Typical actions include repartition, migrate, remap, resize memory tier, adjust admission threshold, trigger compaction, or change retention window. Idempotence and bounded rollback matter because control loops may reissue actions under noisy telemetry.

Recent retrieval and retention lines suggest two additional actuation classes deserve first-class treatment. One is shard-exposure control: the runtime may need to mark a partially refreshed retrieval shard as queryable only for internal warmup, delayed publication, or restricted traffic classes. The other is tier-demotion control: under budget contraction, the runtime may need to summarize, compress, or temporarily demote retention objects instead of making an immediate drop-or-keep decision. These are not implementation details; they are policy-visible actions that materially change user-facing quality.

Systems that expose only monolithic reconfiguration APIs tend to react too slowly under real-time pressure. Fine-grained actuation paths improve responsiveness but require careful conflict handling when multiple policies target overlapping state objects. In practice, this means actuation needs capability descriptors and scope boundaries, so the runtime can tell whether two commands commute, one dominates the other, or both must be serialized through a stronger safety check.

5.4.5 Safety and Recovery Envelope. The literature also suggests that a stateful system remains analytically incomplete without an explicit safety envelope: invariants that must hold during and after control actions. Examples include bounded replay lag, consistency constraints on shared indexes, or no-eviction sets during critical decode windows. Recovery is strongest when integrated into the same envelope rather than treated as a separate mode.

In retrieval and retention settings, two invariants are especially important. First, freshness exposure contracts should specify when a shard may be served despite ongoing rebuild, compaction, or deletion repair. Second, retention demotion contracts should specify which memories are compressible, summarizable, or protected when effective budget drops suddenly. Without these invariants, systems can satisfy local maintenance logic while still violating user-visible correctness or quality assumptions.

The survey literature repeatedly suggests that performance and recovery cannot be cleanly separated once state is dynamic [24, 190, 270]. Any credible blueprint must therefore be judged against both steady-state and recovery invariants. Recovery experiments should reuse the same typed state objects, action boundaries, and exposure contracts as steady-state control; otherwise the system effectively runs two different semantics and the blueprint no longer closes the loop.

This mapping also clarifies why the case studies that follow are intentionally operational rather than paper-by-paper. Each case is a stress test for whether the blueprint closes one or more anti-patterns under a concrete service boundary. The cases are therefore not examples appended after the real argument; they are the first place where the survey's design claims become falsifiable.

5.5 Cross-Domain Case Studies

To make the framework concrete, we examine three representative scenarios in which the blueprint above becomes operational rather than descriptive. The cases are not proposed systems; they show how the same analytical scaffold exposes different control seams once state, disturbance,

Table 8. Mapping recurring anti-patterns to the blueprint contracts that repair them.

Anti-pattern	Missing contract	Primary blueprint layer	Expected design effect
Telemetry-action mismatch	signals must name a legal actuation target and its delay/confidence semantics	observation and disturbance detection	dashboards become control inputs rather than passive monitoring
Policy layering without conflict semantics	hard constraints, soft objectives, and tie-breakers must be ordered explicitly	boundary model and decision kernel	overlapping controllers stop oscillating under burst or skew
One-shot benchmarking of non-stationary mechanisms	disturbance phases and re-stabilization windows must be part of the contract	observation plus safety and recovery envelope	reported gains remain meaningful outside stationary replay
Ignoring write amplification in read-optimized designs	mutation, compaction, deletion repair, and maintenance debt must be charged to the same boundary as read gains	state catalog plus actuation interfaces	deferred rebuild or compaction cost becomes visible during design review
Treating state ownership as static	ownership transfer and no-eviction rules must survive migration and recovery	actuation interfaces plus safety and recovery envelope	scale-out and fault handling share one transferable state model

Table 9. Abstract policy skeleton for unified state governance.

Step	Policy skeleton
1	`observe_signals()` and update telemetry confidence for contention, queueing, local freshness, exposure freshness, and retention pressure.
2	`identify_state_objects()` whose boundary is currently active: hot operator shards, KV pages, retrieval indexes, replay buffers, or demotion candidates.
3	`check_hard_constraints()` for SLA, memory limit, recovery envelope, shard-exposure safety, and quality floor before any optimization action.
4	`rank_actions()` over admit, place, migrate, refresh, compact, expose, demote, and evict using expected utility and maintenance debt.
5	`apply_bounded_action()` with explicit rollback or retry semantics when telemetry is delayed or conflicting.
6	`account_outcome()` by charging the decision against tail latency, quality drift, effective budget loss, and long-horizon reuse value.

and service boundary are specified explicitly. Their role is to test whether the blueprint still says something useful once the discussion is forced to choose concrete ownership rules, telemetry signals, and legal actions under a domain-specific workload.

5.5.1 Multi-Tenant LLM Serving Under Burst Arrival. A serving cluster handling mixed prompt lengths, intermittent bursts, and strict p99 targets is governed primarily by short-lived state objects: KV pages, prefix-share indexes, request queues, tenant-specific adapters, and phase-specific occupancy traces. The relevant control surfaces are admission thresholding, phase-aware scheduling, page placement, reuse-window control, transfer scheduling, and reclamation timing. The difficulty

arises because burst arrivals raise admission pressure, which deepens decode-phase queues, expands KV footprint, and ultimately reduces effective batching while degrading tail latency. The literature indicates that this is not simply an allocator problem. vLLM shows that page-granular KV management can reduce fragmentation and expand feasible batching regions, while Sarathi-Serve, DistServe, Splitwise, and FastGen show that the temporal asymmetry between prefill and decode must be treated as a first-class scheduling seam rather than as noise around a uniform request path [6, 79, 115, 172, 278]. The boundary model is therefore straightforward but unforgiving: memory safety and p99 latency remain hard constraints, while stable throughput is the soft objective and fragmentation slope or phase imbalance becomes the disturbance signal.

The same control problem becomes more demanding once tenant multiplexing and cross-node transfer are added. Punica and AlpaServe show that the state being routed is not limited to generic KV memory; it often includes tenant-specific adapters, partition metadata, and multiplexing constraints whose residency rules change interference patterns [30, 126]. Follow-on systems such as Prefill-as-a-Service, SplitZip, and CacheFlow then suggest that successful phase disaggregation may simply move the next bottleneck to transfer and restoration: once prefill and decode are separated, ownership and movement of KV state become explicit control surfaces in their own right [73, 162, 177]. The synthesis point is that burst robustness depends on coherent lifecycle governance over short-lived serving state rather than on any single optimization knob. What the literature still lacks is a control contract that can rank admission, offload, restoration, and tenant-isolation actions against the same service boundary, so that cross-node KV transfer is charged as part of latency protection rather than hidden as a separate systems debt.

5.5.2 Stateful Stream Processing Under Reconfiguration. A stream engine that must scale out during bursts and recover quickly from worker failure is shaped by keyed windows, operator-state shards, migration logs, progress metadata, and checkpoint boundaries. Here the dominant controls are migration grain, trigger policy, replication factor, checkpoint cadence, and replay window. The central coupling path is that reconfiguration changes locality and transfer volume, while recovery policy changes log pressure and replay interference; both feed back into future contention and scheduling stability. The literature makes clear that this is not one problem but two coupled ones: progress semantics determine when state may advance safely, and ownership-transfer mechanisms determine where that state may move without violating the visibility boundary [7, 20, 157]. Megaphone and related migration work show that finer transfer grain can reduce visible pause time, but only when the runtime already has explicit ownership metadata and progress-aware routing [150, 160].

A practical handoff protocol must therefore treat ownership as a versioned lease: state moves with a progress fence, the old owner retains a bounded replay buffer until the new owner acknowledges, and replay remains idempotent across the boundary [19, 150, 160, 270]. None of the cited systems enforces the full protocol end to end; each secures one piece, which is why reconfiguration and recovery are still evaluated as separate modes. The open gap is a single transfer specification that guarantees both correctness and bounded pause across scale-out and recovery.

The neighboring literature also suggests what this protocol should not omit. If ownership is not lease- or epoch-scoped, the runtime cannot say unambiguously who is the only legal writer after handoff; if replay is not tied to a deterministic boundary, failover becomes a fresh conflict-resolution episode instead of continued execution; and if reconstruction cost is ignored, nominally correct reassignment may still violate the service boundary because the new owner comes up too slowly [33, 164, 165, 204]. This is why the survey treats lease, epoch, and replay-fence semantics as part of one ownership-transfer problem rather than as details borrowed opportunistically from adjacent systems.

Recovery further sharpens the same tradeoff rather than creating a separate one. OSM and asynchronous checkpointing showed early that elasticity and fault tolerance should share a common state-transfer substrate instead of duplicating serialization and ownership logic [19, 51]. Follow-on work in transactional streaming makes the disturbance boundary even clearer: MorphStream and fast-recovery designs imply that dependency shape, replay scheduling, and steady-state execution plans must be co-designed because the same state object is being optimized for latency before failure and for convergence after failure [151, 270]. From this perspective, the case study is not about one better migration algorithm. It is about whether the runtime can preserve logical correctness while harmonizing the orthogonal demands of fine-grained elasticity and parallel recovery under the same ownership contract. The unresolved seam is therefore a disturbance-invariant ownership model: until progress tracking, migration authority, and replay damping are expressed through one legal transfer protocol, stream runtimes will keep solving elasticity and recovery as neighboring but still partially incompatible control loops.

5.5.3 Retrieval Memory Under Continuous Corpus Drift. A RAG service ingesting frequent content updates is governed by vector-index shards, late-interaction caches, graph memory, shard-exposure metadata, and related retention state. Its main controls are incremental indexing policy, refresh cadence, compaction schedule, shard-exposure policy, and stale-memory eviction. The coupling path is a classic lifecycle tradeoff: aggressive updates improve freshness, yet index churn can destabilize latency and retrieval quality if rebuild, repair, and exposure are not coordinated. This becomes clearer when static and dynamic memory layers are compared directly. DPR and ColBERT represent a mostly static retrieval regime in which the main online question is how to search a prepared memory efficiently, whereas FreshDiskANN, SPFresh, and deletion-aware dynamic ANN lines move the systems question to maintenance itself by showing that in-place mutation, locality-preserving repair, and deletion handling determine whether freshness can be sustained without repeated global rebuilds [105, 107, 196, 228, 229]. Milvus and Manu add the distributed analogue: background indexing, segment sealing, shard placement, and query routing decide whether partially refreshed state is merely present or actually safe to expose at the service boundary [71, 211].

Planner-mediated retrieval creates a second control loop above index maintenance. Self-RAG, FlowRAG, and CANDOR-Bench show that once corpora and ANN structures evolve continuously, the dominant tradeoff is no longer simply retrieve versus do not retrieve, but whether to query a stale index, pay immediate rebuild or compaction cost, or defer maintenance and absorb future quality drift [12, 213, 259]. Graph- and hierarchy-enhanced memories such as HippoRAG and RAPTOR increase the value of structured long-horizon retrieval, but they also increase maintenance debt in graph rewiring, abstraction refresh, and stale-structure cleanup [189, 218]. These families are comparable because each makes exposure conditions explicit, not because they report one common retrieval metric or maintenance envelope. The key point is that retrieval quality is a lifecycle property rather than a static model property. The missing piece is not another isolated refresh heuristic, but an exposure-aware accounting layer that can turn recall drift, rebuild debt, and planner confidence into one auditable trigger policy for when memory may remain queryable, when it must degrade gracefully, and when maintenance has to preempt service.

6 EVALUATION AND RESEARCH OUTLOOK

The remaining question is evidentiary: how should the field recognize genuine progress once state-management mechanisms interact across layers and over time? This section proposes a disturbance-oriented vocabulary and a concise forward agenda for judging stability, composability, and long-horizon reuse quality under realistic lifecycle stress.

6.1 Evaluation Dimensions for Future Surveys and Systems

An enduring weakness of the area is evaluation fragmentation. Access papers often report throughput and contention reduction, execution papers report speedup or energy savings, and evolution papers report quality or forgetting. These are all important, but they make it hard to compare whether two systems are solving the same problem at different phases of the loop or genuinely addressing different control surfaces. A more coherent evaluation vocabulary should at least track four dimensions.

The problem is not only that metrics differ, but that each literature often leaves one adjacent boundary weakly audited. Stream and transactional papers are usually strongest on throughput, contention, or recovery latency, yet often separate disturbance phases that should be evaluated as one control loop. Serving papers increasingly report TTFT, TPOT, or goodput, but many still under-specify when reuse, transfer, or restoration remain semantically valid under mixed tenants and mixed phases. Retrieval and retention papers are typically strongest on accuracy-, recall-, or forgetting-facing outcomes, yet much weaker on maintenance debt, exposure safety, and long-horizon operational cost once compaction, rebuild, or budget contraction begins. Put differently, the field does not merely lack a common metric table; it lacks a stable habit of checking whether a local gain survives the neighboring boundary that makes the mechanism operationally meaningful.

The first dimension is *steady-state efficiency*, including throughput, amortized cost per useful result, and locality-sensitive resource use. The second is *tail behavior*, including percentile latency, burst amplification, and recovery delay. The third is *state quality*, including bounded approximation error, retrieval quality under drift, and retention value under memory budgets. The fourth is *operational sustainability*, including migration cost, update overhead, and how quickly a control loop re-stabilizes after workload or hardware changes. Systems that score well on only one dimension can still be valuable, but they should be described as local improvements rather than full state-management solutions.

At this level of comparison, the four-part vocabulary should be interpreted as an evidence ladder rather than as a menu of interchangeable metrics. Consistent with the evidence posture in Section 2, a strong paper first makes its state object and control boundary explicit, then demonstrates local mechanism gains, then shows that these gains survive disturbance, and finally tests whether they remain meaningful when composed with neighboring controllers or maintenance tasks. Many influential systems papers stop one rung earlier: they establish an important mechanism but do not yet show that the mechanism closes the full runtime loop. This is why foundational mechanisms, transfer demonstrations, and frontier lifecycle warnings should not be flattened into identical evidence of maturity even when all three report positive local results.

Figure 7 turns this vocabulary into a compact evaluation protocol. The key shift is from one-shot benchmarking to multi-phase disturbance testing: warmup, burst, reconfiguration, drift, and long-horizon maintenance should be treated as one connected experiment rather than separate appendices. Only then can a paper claim that a controller remains useful across the lifecycle regime in which the state object actually matters.

This evaluation vocabulary compares systems without flattening their contributions. A streaming scheduler and a dynamic retriever may report different headline metrics, yet both can be judged by whether they expose a stable state object, define an optimization boundary around it, and connect local policy decisions to long-horizon system behavior. This comparability underscores the utility of the state-management perspective beyond any single subcommunity. More importantly, it creates a shared standard for saying that two papers are complementary, incomparable, or genuinely cumulative instead of treating every positive result as evidence of equivalent maturity.

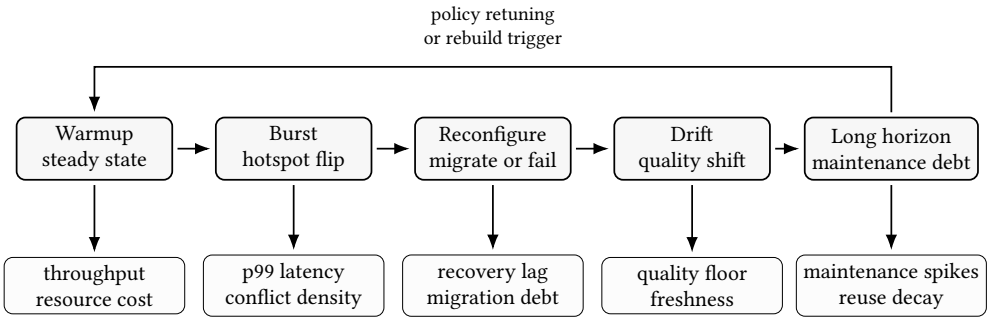

Fig. 7. A disturbance-oriented evaluation protocol for stateful systems. The same mechanism should be judged across steady-state, burst, reconfiguration, drift, and long-horizon maintenance phases rather than only on a single stationary run.

6.2 An Integrated Research Agenda

The design implications above can be distilled into four forward-looking pressure points for stateful infrastructure research. They are not a generic wishlist. Each follows from a recurring unresolved seam in the preceding sections: fragmented observability, cross-tier scheduling without stable ownership contracts, memory services without portable lifecycle semantics, and evaluation practices that hide disturbance and composition cost. Together they shift the discussion from local mechanism choice toward the missing contracts and shared control layers that the literature repeatedly gestures toward but has not yet stabilized.

6.2.1 Unified State Observability. The first pressure point is a telemetry model that spans more than one state family. Today, most systems expose only the slice their local mechanism needs: stream runtimes report queue or frontier signals, serving systems report cache occupancy and queue depth, retrieval stacks report freshness or recall proxies, and retention systems report task accuracy or memory budget. The field still lacks an observability contract that can express state identity, lifecycle stage, confidence, delay, and exposure semantics across these domains, allowing a runtime to decide whether a hotspot flip in streaming, a phase-imbalance spike in serving, and a freshness lag in retrieval are instances of the same disturbance class or fundamentally different control events. The issue is not limited to monitoring. Without such a shared telemetry contract, multi-controller systems cannot reliably compose decisions because each policy reasons over a partial and differently delayed view of state. From the literature's current trajectory, unified state observability increasingly looks like a prerequisite for reusable policy composition, disturbance-aware benchmarking, and portable runtime-control APIs. The harder requirement is a telemetry language rich enough to say not only that something is changing, but also which lifecycle boundary is moving, how trustworthy that signal is, and which controller is allowed to act on it.

6.2.2 State-Centric Scheduling on Heterogeneous Hardware. The second pressure point is scheduling that treats state placement, access topology, and quality boundary as one optimization problem on heterogeneous hardware. Existing literature already shows pieces of this idea: NUMA-aware stream scheduling, CPU/GPU-aware state paths, paged KV management, and offloaded memory layers each improve one slice of the cost surface [115, 195, 249, 257]. The unresolved requirement is a scheduler that can jointly reason about who touches state, where that state should reside, how expensive it is to move or compress, and whether mutation or refresh is occurring concurrently.

This pressure point matters most for modern AI services because they combine short-lived and long-lived state on the same heterogeneous substrate. A scheduler that optimizes decode latency while ignoring retrieval-refresh traffic, or that optimizes vector maintenance while ignoring serving interference, will continue to converge on local optima that fail under mixed workloads. The literature therefore points toward state-centric scheduling policies that can be reasoned about across GPUs, CPUs, SSD tiers, and network-separated pools rather than only within one device island. In other words, the scheduler must reason over cross-tier state ownership and movement budgets directly, not merely infer them as side effects of task placement.

6.2.3 Memory Middleware for Dynamic AI Services. Retrieval-augmented generation, continual learning, and agentic services increasingly point toward a memory middleware layer that does more than store reusable artifacts. Such a layer would need to make memory state observable, expose actuation points over update and reuse, and arbitrate among heterogeneous state objects that age on different timelines. The same middleware perspective likely extends to short-lived serving state such as KV caches and prefix-sharing indices, because these objects already demand explicit admission, sharing, and reclamation policies. Existing systems provide early building blocks, and recent proposals such as Neuromem and SAGE make the middleware seam itself more explicit, but a stable general abstraction is still missing [138, 255].

The current literature is most informative when read by which missing middleware responsibility each paper surfaces. One line exposes maintenance pressure: evolving retrievers and mutating ANN substrates force the runtime to choose between immediate freshness cost and future quality drift [213, 259]. A second line exposes lifecycle observability by decomposing insertion, consolidation, retrieval, and integration into auditable stages rather than treating memory as one opaque module [255]. A third line exposes actuation by turning vector indexes, windows, joins, and asynchronous updates into pluggable reasoning services that can be orchestrated explicitly [138]. Read together at that responsibility level, these papers suggest early maintenance mechanisms, early lifecycle telemetry, and early workflow-native execution interfaces; what they still do not provide is the policy layer that can arbitrate among them once they coexist inside one service boundary.

That missing layer matters because future middleware will have to span multiple concrete state forms simultaneously: replicated key-value layers for durable control state, vector and late-interaction indexes for semantic retrieval, compressed approximate-nearest-neighbor structures for efficient memory search, and attention-state managers for short-lived per-request reuse [26, 37, 39, 93, 105, 107, 115, 149]. The architectural problem is therefore not building a larger cache, but defining common lifecycle verbs, ownership rules, and maintenance-accounting semantics across state objects with incompatible lifetimes, refresh costs, and quality consequences. Until runtimes can specify how shard exposure, replay eligibility, retrieval freshness, KV transfer, and budget demotion coexist inside one policy regime, memory middleware will remain a cluster of strong local mechanisms rather than a stable infrastructure abstraction.

6.2.4 End-to-End Evaluation Beyond Local Speedups. The fourth pressure point is evaluation that treats local mechanisms as candidates for infrastructural reuse rather than as isolated performance tricks. The field already knows how to produce strong micro-results for concurrency control, caching, approximation, vector indexing, or replay heuristics. What it lacks is systematic evidence that these mechanisms remain useful when combined with other controllers and judged against service-level boundaries such as p99 latency, failure recovery, freshness exposure, effective retention budget, and long-horizon reuse quality. Disturbance-driven protocols therefore matter not as an optional appendix, but as the methodology needed to determine whether a policy improves the runtime or merely shifts debt outside the measurement window.

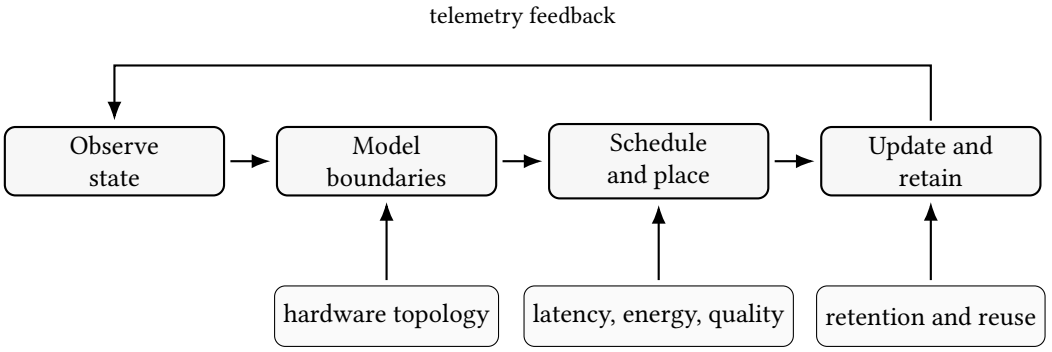

Fig. 8. Integrated runtime loop for future stateful infrastructure. Observation, boundary modeling, scheduling, and update control should be coupled rather than deployed as isolated policies.

This agenda also demands stronger negative evidence. Future evaluations should report sign reversals, maintenance spikes, policy conflicts, and degradation under mixed disturbance rather than only their best stable regime. Without that level of accounting, state-management claims will continue to look stronger in isolation than they are in deployment. A mature infrastructure literature should make it normal to ask not only when a mechanism works, but also which boundary conditions reliably cause it to stop composing with the rest of the runtime.

The broader implication is that the field no longer mainly lacks mechanisms; it lacks comparably audited evidence about when those mechanisms transfer across domains. Comparative work should therefore keep separating reusable abstractions from regime-specific wins. The more systems expose ownership, lifecycle stage, disturbance regime, and service boundary in comparable terms, the more likely their mechanisms are to survive beyond the benchmark niche that introduced them.

Figure 8 condenses this agenda into one runtime picture. The unifying requirement is not to invent a universal optimizer, but to make state transitions observable enough that multiple optimizers can safely cooperate. In practice, a stateful runtime should know what it is storing, why that state matters, how expensive it is to move or mutate, and what downstream service boundary it affects.

7 CONCLUSION

State management is no longer well described as a secondary storage concern alone; for many modern services it is increasingly a runtime control problem that determines whether the system remains fast, stable, and reusable under load, disturbance, and change. The value of that frame is not rhetorical breadth but comparative discipline: it makes streaming, serving, retrieval, and retention results legible as answers to related governance problems without claiming that their service boundaries or correctness conditions are interchangeable. The literature already provides many of the necessary ingredients, but not yet a stable integration layer. Access-control work established the observation-model-control chain for shared state; hardware-conscious and quality-aware execution work showed why local speedups fail without boundary modeling; and retrieval and retention systems made lifecycle debt, freshness, and reuse explicit runtime concerns rather than background maintenance tasks. What remains underdeveloped is a shared substrate that can expose typed state catalogs, enforce precedence among overlapping policies, account for maintenance debt, and preserve service-level guarantees while memory objects change lifetime, representation, ownership, and exposure conditions over time. For the systems emphasized

here, the more informative bar for future work is contract-level rather than component-level: whether runtimes can keep exposure freshness auditable across shards, keep retention budgets interpretable under elasticity, and keep maintenance from leaking unpredictably into user-visible behavior.

The same contract-level bar should also discipline the survey literature itself. A mature synthesis cannot rely on citation breadth alone; it should separate recurring control abstractions from regime-specific wins and distinguish local mechanism results from evidence that survives disturbance, controller interaction, and service-boundary scrutiny. Under that standard, efficient state management is best treated as an organizing systems question about how runtimes expose, govern, and defend reusable state across disturbance, heterogeneity, and time. If the area converges, it is more likely to do so around shared contracts for observing, scheduling, evolving, and safeguarding state than around one dominant mechanism family.

REFERENCES

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Conference on Innovative Data Systems Research*.

[2] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (2003), 120–139.

[3] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J. Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: KV Cache Reduction through Key Tokens Selection for Efficient Generative Inference. *arXiv preprint arXiv:2403.09054* (2024).

[4] Saurabh Agarwal, Bodun Hu, Anyong Mao, Aditya Akella, and Shivaram Venkataraman. 2026. SYMPHONY: Enabling Compute-Memory Disaggregation in LLM Serving Systems. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*.

[5] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. *Proceedings of the ACM European Conference on Computer Systems* (2013).

[6] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. *arXiv preprint arXiv:2403.02310* (2024).

[7] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[8] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernandez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[9] Rahaf Aljundi, Min Lin, Baptiste Goujaud, and Yoshua Bengio. 2019. Online Continual Learning with Maximally Interfered Retrieval. In *Advances in Neural Information Processing Systems*.

[10] Daiyaan Arfeen, Zhen Zhang, Xinwei Fu, Gregory R. Ganger, and Yida Wang. 2024. PipeFill: Using GPUs During Bubbles in Pipeline-parallel LLM Training. *arXiv preprint arXiv:2410.07192* (2024).

[11] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2018).

[12] Akari Asai, Sewon Min, Zexuan Zhong, Danqi Chen, Hannaneh Hajishirzi, Wen-tau Yih, and Luke Zettlemoyer. 2024. Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. *Proceedings of the International Conference on Learning Representations* (2024).

[13] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data Systems Research*.

[14] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Angela Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Laurent Sifre, Oriol Vinyals, et al. 2022. Improving Language Models by Retrieving from Trillions of Tokens. *Proceedings of the International Conference on Machine Learning* (2022).

[15] Pietro Buzzega, Matteo Boschini, Angelo Porrello, Davide Abati, and Simone Calderara. 2020. Dark Experience for General Continual Learning: A Strong, Simple Baseline. In *Advances in Neural Information Processing Systems*.

[16] Weilin Cai, Le Qin, and Jiayi Huang. 2025. MoC-System: Efficient Fault Tolerance for Sparse Mixture-of-Experts Model Training. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[17] Shiyi Cao, Shu Liu, Tyler Griggs, Peter Schafhalter, Xiaoxuan Liu, Ying Sheng, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. 2025. MoE-Lightning: High-Throughput MoE Inference on Memory-constrained GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. https://doi.org/10.1145/3669940.3707267

[18] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*.

[19] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2015. Lightweight Asynchronous Snapshots for Distributed Dataflows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[21] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2010. SEEP: Scalable and Elastic Event Processing. In *Proceedings of the International Middleware Conference*.

[22] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[23] Badrish Chandramouli, Jonathan Goldstein, Songyun Duan, Kai Ma, Ann McCloskey, Vivek Narasayya, Rajeev Ramamurthy, Frederick Reiss, Bogdan Saftoiu, Shi Shen, and Abhishek Tebbi. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[24] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems* 3, 1 (1985), 63–75.

[25] Chia-Hao Chang, Vikram Sharma Mailthody, Jihoon Han, Zaid Qureshi, Anand Sivasubramaniam, and Wen-Mei Hwu. 2024. GMT: GPU Orchestrated Memory Tiering for the Big Data Era. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems* 26, 2 (2008).

[27] Arslan Chaudhry, Marc'Aurelio Ranzato, Marcus Rohrbach, and Mohamed Elhoseiny. 2019. Efficient Lifelong Learning with A-GEM. In *Proceedings of the International Conference on Learning Representations*.

[28] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *Proceedings of the VLDB Endowment* (2024).

[29] Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, and Haibo Chen. 2025. Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. https://doi.org/10.1145/3731569.3764808

[30] Lequn Chen, Yankai Li, Qinghao Zhang, Ion Stoica, Joseph E. Gonzalez, and Matei Zaharia. 2024. Punica: Multi-Tenant LoRA Serving. *Proceedings of Machine Learning and Systems* (2024).

[31] Lei Chen, Shuhao Zhang, and Bingsheng He. 2018. IncApprox: A Data Analytics System for Incremental Approximate Computing. *Proceedings of the VLDB Endowment* (2018).

[32] Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. 2024. COAXIAL: A CXL-Centric Memory System for Scalable Servers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 24)*.

[33] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi

Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2012).

[34] Daniel Crankshaw, Ratik Mahajan, Xin Wang, Guilio Zhou, Zhijie Wen, Aakanksha Gokhale, Nitika Dey, Nikhil Agarwal, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2020. InferLine: ML Prediction Pipeline Provisioning and Management for Tight Latency Objectives. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

[35] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

[36] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *Comput. Surveys* 44, 3 (2012).

[37] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *Proceedings of the International Conference on Learning Representations*.

[38] Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. 2021. A Continual Learning Survey: Defying Forgetting in Classification Tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 7 (2021), 3366–3385.

[39] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

[40] David Domingo and Sudarsun Kannan. 2021. pFSCK: Accelerating File System Checking and Repair for Modern Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*.

[41] Juechu Dong, Jonah Rosenblum, and Satish Narayanasamy. 2024. Toleo: Scaling Freshness to Tera-scale Memory Using CXL and PIM. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. https://doi.org/10.1145/3622781.3674180

[42] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

[43] Kuntai Du, Bowen Wang, Chen Zhang, Yiming Cheng, Qing Lan, Hejian Sang, Yihua Cheng, Jiayi Yao, Xiaoxuan Liu, Yifan Qiao, Ion Stoica, and Junchen Jiang. 2025. PrefillOnly: An Inference Engine for Prefill-only Workloads in Large Language Model Applications. *arXiv preprint arXiv:2505.07203* (2025).

[44] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: A Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.

[45] Ruibo Fan, Xiangrui Yu, Peijie Dong, Zeyu Li, Gu Gong, Qiang Wang, Wei Wang, and Xiaowen Chu. 2025. SpInfer: Leveraging Low-Level Sparsity for Efficient Large Language Model Inference on GPUs. In *Proceedings of the European Conference on Computer Systems*.

[46] Chao Fang, Man Shi, Robin Geens, Arne Symons, Zhongfeng Wang, and Marian Verhelst. 2025. Anda: Unlocking Efficient LLM Inference with a Variable-Length Grouped Activation Data Format. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA61900.2025.00110

[47] Fei Fang, Yifan Hua, Shengze Wang, Ruilin Zhou, Yi Liu, Chen Qian, and Xiaoxue Zhang. 2026. PlanetServe: A Decentralized, Scalable, and Privacy-Preserving Overlay for Democratizing Large Language Model Serving. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*.

[48] Marco Federici, Davide Belli, Mart van Baalen, Amir Jalalirad, Andrii Skliar, Bence Major, Markus Nagel, and Paul Whatmough. 2025. Efficient LLM Inference using Dynamic Input Pruning and Cache-Aware Masking. In *Proceedings of Machine Learning and Systems*, Vol. 7.

[49] Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. 2025. WindServe: Efficient Phase-Disaggregated LLM Serving with Stream-based Dynamic Scheduling. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. https://doi.org/10.1145/3695053.3730999

[50] Weiqi Feng, Yangrui Chen, Shaoyu Wang, Yanghua Peng, Haibin Lin, and Minlan Yu. 2025. Optimus: Accelerating Large-Scale Multi-Modal LLM Training by Bubble Exploitation. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[51] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[52] Xiang Fu, Weiping Zhang, Xin Huang, Wubiao Xu, Shiman Meng, Luanzheng Guo, and Kento Sato. 2024. AutoCheck: Automatically Identifying Variables for Checkpointing by Data Dependency Analysis. In *IEEE International Parallel*

and Distributed Processing Symposium (IPDPS 24).

[53] Yaosheng Fu, Evgeny Bolotin, Aamer Jaleel, Gal Dalal, Shie Mannor, Jacob Subag, Noam Korem, Michael Behar, and David Nellans. 2024. AutoScratch: ML-Optimized Cache Management for Inference-Oriented GPUs. In *Proceedings of Machine Learning and Systems*.

[54] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. 2022. SFS: Smart OS Scheduling for Serverless Functions. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1109/SC41404.2022.00047

[55] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.

[56] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. 2025. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[57] Hongru Gao, Shuhao Zhang, Xiaofei Liao, and Hai Jin. 2026. GRACE: Alleviating Reconstruction Cost in Dynamic Graph Processing Systems. In *Proceedings of the IEEE International Conference on Data Engineering*.

[58] Shiwei Gao, Youmin Chen, and Jiwu Shu. 2025. Fast State Restoration in LLM Serving with HCache. In *Proceedings of the European Conference on Computer Systems*.

[59] Shiwei Gao, Qing Wang, Shaoxun Zeng, Youyou Lu, and Jiwu Shu. 2025. Weaver: Efficient Multi-LLM Serving with Attention Offloading. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[60] Xin Gao, Sibasish Acharya, Sihui Han, Yongxiong Ren, Yanli Zhao, Liang Luo, Chucheng Wang, Pradeep Fernando, Saurabh Mishra, Siqi Yan, Yicong Du, Elzbieta Krepska, Intaik Park, Min Ni, Qunshu Zhang, and Shen Li. 2025. DECK: Experiences on Delta Checkpointing for Industrial Recommendation Systems. *Proceedings of the VLDB Endowment* (2025).

[61] Gerasimos Gerogiannis, Stijn Eyerman, Evangelos Georganas, Wim Heirman, and Josep Torrellas. 2025. DECA: A Near-Core LLM Decompression Accelerator Grounded on a 3D Roofline Model. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*.

[62] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *Proceedings of Machine Learning and Systems*.

[63] In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. 2025. Pie: A Programmable Serving System for Emerging LLM Applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*.

[64] Imanol Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[65] Ruihao Gong, Shihao Bai, Siyu Wu, Yunqian Fan, Zaijun Wang, Xiuhong Li, Hailong Yang, and Xianglong Liu. 2025. Past-Future Scheduler for LLM Serving under SLA Guarantees. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[66] Yufeng Gu, Alireza Khadem, Sumanth Umesh, Ning Liang, Xavier Servot, Onur Mutlu, Ravi Iyer, and Reetuparna Das. 2025. PIM Is All You Need: A CXL-Enabled GPU-Free System for Large Language Model Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.

[67] Yue Guan, Xinwei Qiang, Zaifeng Pan, Daniels Johnson, Yuanwei Fang, Keren Zhou, Yuke Wang, Wanlu Li, Yufei Ding, and Adnan Aziz. 2025. Mercury: Unlocking Multi-GPU Operator Optimization for LLMs via Remote Memory Scheduling. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*.

[68] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

[69] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Patrick Valduriez, and Claudio Soriente. 2012. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (2012), 2351–2365.

[70] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.

[71] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: A Cloud Native Vector Database Management System. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3548–3561. https://doi.org/10.14778/3554821.3554843

[72] Tianyu Guo, Zhiguang Chen, Xianwei Zhang, Nong Xiao, Jiangsu Du, and Yutong Lu. 2025. gLLM: Global Balanced Pipeline Parallelism Systems for Distributed LLMs Serving with Token Throttling. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 25)*.

[73] Yipin Guo and Siddharth Joshi. 2026. SplitZip: Ultra Fast Lossless KV Compression for Disaggregated LLM Serving. *arXiv preprint arXiv:2605.01708* (2026).

[74] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. REALM: Retrieval-Augmented Language Model Pre-Training. *Proceedings of the International Conference on Machine Learning* (2020).

[75] Tyler L. Hayes, Nathan D. Cahill, and Christopher Kanan. 2020. REMIND: Real-Time Continual Learning without Catastrophic Forgetting. In *Proceedings of the European Conference on Computer Vision*.

[76] Congjie He, Yeqi Huang, Pei Mu, Ziming Miao, Jilong Xue, Lingxiao Ma, Fan Yang, and Luo Mai. 2025. WaferLLM: Large Language Model Inference at Wafer Scale. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[77] Yongjun He, Haofeng Yang, Yao Lu, Ana Klimovic, and Gustavo Alonso. 2025. Resource Multiplexing in Tuning and Serving Large Language Models. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[78] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *Comput. Surveys* 46, 4 (2014).

[79] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. 2024. DeepSpeed-FastGen: High-Throughput Text Generation for LLMs via MII and DeepSpeed-Inference. *arXiv preprint arXiv:2401.08671* (2024).

[80] Jeongmin Hong, Sungjun Cho, Geonwoo Park, Wonhyuk Yang, Young-Ho Gong, and Gwangsun Kim. 2024. Bandwidth-Effective DRAM Cache for GPUs with Storage-Class Memory. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA 2024)*.

[81] Ke Hong, Xiuhong Li, Lufang Chen, Qiuli Mao, Guohao Dai, Xuefei Ning, Shengen Yan, Yun Liang, and Yu Wang. 2025. SOLA: Optimizing SLO Attainment for Large Language Model Serving with State-Aware Scheduling. In *Proceedings of Machine Learning and Systems*.

[82] Yuxuan Hu, Shuhao Zhang, and Bingsheng He. 2018. StreamApprox: Approximate Computing for Stream Analytics. *IEEE Transactions on Knowledge and Data Engineering* (2018).

[83] Zhisheng Hu, Pengfei Zuo, Yizou Chen, Chao Wang, Junliang Hu, and Ming-Chang Yang. 2024. Aceso: Achieving Efficient Fault Tolerance in Memory-Disaggregated Key-Value Stores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.

[84] Chunyue Huang, Shuang Liu, Xinyi Zhang, Wenhao Li, Wei Lu, and Xiaoyong Du. 2025. Chimera: Mitigating Ownership Transfers in Multi-Primary Shared-Storage Cloud-Native Databases. *Proceedings of the VLDB Endowment* (2025).

[85] Yibo Huang, Haowei Chen, Newton Ni, Yan Sun, Vijay Chidambaram, Dixin Tang, and Emmett Witchel. 2025. Tigon: A Distributed Database for a CXL Pod. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[86] Yuzhou Huang, Yapeng Jiang, Zicong Hong, Wuhui Chen, Bin Wang, Weixi Zhu, Yue Yu, and Zibin Zheng. 2025. Obscura: Concealing Recomputation Overhead in Training of Large Language Models with Bubble-filling Pipeline Transformation. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[87] Zimeng Huang, Hao Nie, Haonan Jia, Bo Jiang, Junchen Guo, Jianyuan Lu, Rong Wen, Biao Lyu, Shunmin Zhu, and Xinbing Wang. 2025. FlowCheck: Decoupling Checkpointing and Training of Large-Scale Models. In *Proceedings of the European Conference on Computer Systems*.

[88] Nikoleta Iliakopoulou, Jovan Stojkovic, Chloe Alverti, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2024. Chameleon: Adaptive Caching and Scheduling for Many-Adapter LLM Inference Environments. *arXiv preprint arXiv:2411.17741* (2024).

[89] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the ACM European Conference on Computer Systems*.

[90] Gautier Izacard, Edouard Grave, Mathilde Caron, Herve Jegou, et al. 2022. Atlas: Few-Shot Learning with Retrieval Augmented Language Models. *Journal of Machine Learning Research* 24 (2022).

[91] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova. 2024. ExtMem: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*.

[92] Hongsun Jang, Jaeyong Song, Jaewon Jung, Jaeyoung Park, Youngsok Kim, and Jinho Lee. 2024. Smart-Infinity: Fast Large Language Model Training using Near-Storage Processing on a Real System. In *2024 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA57654.2024.00034

[93] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128.

[94] Keshav Vinayak Jha, Shweta Pandey, Murali Annavaram, and Arkaprava Basu. 2025. HyCache: Hybrid Caching for Accelerating DNN Input Preprocessing Pipelines. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[95] Houxiang Ji, Yifan Yuan, Yang Zhou, Ipoom Jeong, Ren Wang, Saksham Agarwal, and Nam Sung Kim. 2025. Re-architecting End-host Networking with CXL: Coherence, Memory, and Offloading. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*.

[96] Sheng Jiang and Ming Liu. 2025. Building an Elastic Block Storage over EBOFs Using Shadow Views. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*.

[97] Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. 2025. NEO: Saving GPU Memory Crisis with CPU Offloading for Online LLM Inference. In *Proceedings of Machine Learning and Systems*.

[98] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Taiyi Wang, Bin Cui, Ana Klimovic, and Eiko Yoneki. 2025. ThunderServe: High-performance and Cost-efficient LLM Serving in Cloud Environments. In *Proceedings of the Eighth Conference on Machine Learning and Systems (MLSys 2025)*.

[99] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*.

[100] Sian Jin, Sheng Di, Frederic Vivien, Daoce Wang, Yves Robert, Dingwen Tao, and Franck Cappello. 2024. Concealing Compression-accelerated I/O for HPC Applications through In Situ Task Scheduling. In *Proceedings of the Nineteenth European Conference on Computer Systems*.

[101] Donghyeon Joo, Helya Hosseini, Ramyad Hadidi, and Bahar Asgari. 2025. Coruscant: Co-Designing GPU Kernel and Sparse Tensor Core to Advocate Unstructured Sparsity in Efficient LLM Inference. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. https://doi.org/10.1145/3725843.3756065

[102] Robert Kallman, Hideaki Kimura, Justin Natkins, Andrew Pavlo, Alexander Rasin, Stan Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *Proceedings of the VLDB Endowment*, Vol. 1. 1496–1499.

[103] Aditya K. Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. POD-Attention: Unlocking Full Prefill-Decode Overlap for Faster LLM Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. https://doi.org/10.1145/3676641.3715996

[104] Hao Kang, Srikant Bharadwaj, James Hensman, Tushar Krishna, Victor Ruhle, and Saravan Rajmohan. 2025. TurboAttention: Efficient Attention Approximation for High Throughputs LLM. In *Proceedings of Machine Learning and Systems*.

[105] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

[106] Ahmad Faraz Khan, Samuel Fountain, Ahmed M. Abdelmoniem, Ali R. Butt, and Ali Anwar. 2025. FLStore: Efficient Federated Learning Storage for Non-training Workloads. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[107] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*.

[108] Hyungyo Kim, Jinghan Huang, Nachuan Wang, Amir Yazdanbakhsh, Qirong Xia, and Nam Sung Kim. 2025. LIA: A Single-GPU LLM Inference Acceleration with Cooperative AMX-Enabled CPU-GPU Computation and CXL Offloading. In *Proceedings of the International Symposium on Computer Architecture*.

[109] Minsu Kim, Seongmin Hong, RyeoWook Ko, Soongyu Choi, Hunjong Lee, Junsoo Kim, Joo-Young Kim, and Jongse Park. 2025. Oaken: Fast and Efficient LLM Serving with Online-Offline Hybrid KV Cache Quantization. In *Proceedings of the International Symposium on Computer Architecture*.

[110] Wonung Kim, Yubin Lee, Yoonsung Kim, Jinwoo Hwang, Seongryong Oh, Jiyong Jung, Aziz Huseynov, Woong Gyu Park, Chang Hyun Park, Divya Mahajan, and Jongse Park. 2025. Pimba: A Processing-in-Memory Acceleration for Post-Transformer Large Language Model Serving. In *Proceedings of the 58th Annual IEEE/ACM International Symposium on Microarchitecture*.

[111] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming Catastrophic Forgetting

in Neural Networks. *Proceedings of the National Academy of Sciences* 114, 13 (2017), 3521–3526.

[112] Abhishek Vijaya Kumar, Gianni Antichi, and Rachee Singh. 2025. Aqa: Network-Accelerated Memory Offloading for LLMs in Scale-Up GPU Domains. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[113] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*.

[114] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. 2020. FVM: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[115] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. *arXiv preprint arXiv:2309.06180* (2023).

[116] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[117] Hyojung Lee, Daehyeon Baek, Jimyoung Son, Jieun Choi, Kihyo Moon, and Minsung Jang. 2025. PAISE: PIM-Accelerated Inference Scheduling Engine for Transformer-based LLM. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. https://doi.org/10.1109/HPCA61900.2025.00126

[118] Hwanjun Lee, Minho Kim, Yeji Jung, Seonmu Oh, Ki-Dong Kang, Seunghak Lee, and Daehoon Kim. 2025. Beyond Page Migration: Enhancing Tiered Memory Performance via Integrated Last-Level Cache Management and Page Migration. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO 58)*.

[119] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 155–172. https://www.usenix.org/conference/osdi24/presentation/lee

[120] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich K"uttler, Mike Lewis, Wen-tau Yih, Tim Rockt"aschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems* (2020).

[121] Changwu Li, Tongjun Shi, Shuhao Zhang, Binbin Chen, Bingsheng He, Xiaofei Liao, and Hai Jin. 2026. StreamFP: Fingerprint-guided Data Selection for Efficient Stream Learning. In *Proceedings of the ACM Web Conference*.

[122] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[123] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. 2025. Toppings: CPU-Assisted, Rank-Aware Adapter Serving for LLM Inference. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[124] Yading Li, Dandan Song, Yuhang Tian, Hao Wang, Changzhi Zhou, and Shuhao Zhang. 2025. A Framework of Knowledge Graph-Enhanced Large Language Model Based on Global Planning. *IEEE Transactions on Knowledge and Data Engineering* 38, 2 (2025).

[125] Yading Li, Dandan Song, Changzhi Zhou, Yuhang Tian, Hao Wang, Ziyi Yang, and Shuhao Zhang. 2024. A Framework of Knowledge Graph-Enhanced Large Language Model Based on Question Decomposition and Atomic Retrieval. In *Findings of the Association for Computational Linguistics: EMNLP*.

[126] Zhuohan Li, Honghao Lin, Hao Zhang, and et al. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2023).

[127] Zhouyang Li, Yuliang Liu, Wei Zhang, Tailing Yuan, Bin Chen, and Chengru Song. 2025. SlimPipe: Memory-Thrifty and Efficient Pipeline Parallelism for Long-Context LLM Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/3712285.3759855

[128] Zhiyue Li and Guangyan Zhang. 2024. StreamCache: Revisiting Page Cache for File Scanning on Fast Storage Devices. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*.

[129] Dimitrios Liakopoulos, Prasoon Sinha, Tianrui Hu, Myungjin Lee, and Neeraja J. Yadwadkar. 2025. MaverIQ: Fingerprint-Guided Extrapolation and Fragmentation-Aware Layering for Intent-Based LLM Serving. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 25)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3712285.3759867

[130] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. 2025. Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN Training with Reconfigurable Parallelism. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[131] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU Memory Sharing for Concurrent DNN Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[132] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 929–945.

[133] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, Xin Liu, Aurojit Panda, and Jinyang Li. 2025. Understanding Stragglers in Large Model Training Using What-if Analysis. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[134] Junfeng Lin, Ziming Liu, Yang You, Jun Wang, Weihao Zhang, and Rong Zhao. 2025. WeiPipe: Weight Pipeline Parallelism for Communication-Effective Long-Context Large Model Training. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. https://doi.org/10.1145/3710848.3710869

[135] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2025. QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving. In *Proceedings of Machine Learning and Systems*.

[136] Chaoqiang Liu, Haifeng Liu, Dan Chen, Yu Huang, Yi Zhang, Wenjing Xiao, Xiaofei Liao, and Hai Jin. 2025. HeterRAG: Heterogeneous Processing-in-Memory Acceleration for Retrieval-Augmented Generation. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*.

[137] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered Memory Management Beyond Hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[138] Jun Liu, Peilin Liu, Ruicheng Zhang, Senlei Zhang, Yanbo Chen, Ziao Wang, Jinyun Yang, Mingqi Wang, Shuhao Zhang, Xiaofei Liao, and Hai Jin. 2026. SAGE: A Dataflow-Native Framework for Modular, Controllable, and Transparent LLM-Augmented Reasoning. In *Proceedings of the International Conference on Machine Learning*.

[139] Lian Liu, Shixin Zhao, Bing Li, Haimeng Ren, Zhaohui Xu, Mengdi Wang, Xiaowei Li, Yinhe Han, and Ying Wang. 2025. Make LLM Inference Affordable to Everyone: Augmenting GPU Memory with NDP-DIMM. In *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA 2025)*.

[140] Qianli Liu, Zicong Hong, Peng Li, Fahao Chen, and Song Guo. 2025. MELL: Memory-Efficient Large Language Model Serving via Multi-GPU KV Cache Management. *CoRR* abs/2501.06709 (2025). https://doi.org/10.48550/arXiv.2501.06709

[141] Weijian Liu, Mingzhen Li, Guangming Tan, and Weile Jia. 2025. Mario: Near Zero-cost Activation Checkpointing in Pipeline Parallelism. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. https://doi.org/10.1145/3710848.3710878

[142] Yang Liu, Yunfei Gu, Liqiang Zhang, Chentao Wu, Guangtao Xue, Jie Li, Minyi Guo, Junhao Hu, and Jie Meng. 2026. CacheSlide: Unlocking Cross Position-Aware KV Cache Reuse for Accelerating LLM Serving. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*. USENIX Association. https://www.usenix.org/conference/fast26/presentation/liu-yang

[143] Yuhan Liu, Yuyang Huang, Jiayi Yao, Shaoting Feng, Zhuohan Gu, Kuntai Du, Hanchen Li, Yihua Cheng, Junchen Jiang, Shan Lu, Madan Musuvathi, and Esha Choukse. 2026. DroidSpeak: KV Cache Sharing Across Fine-tuned Model Variants. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*.

[144] Zihan Liu, Xinhao Luo, Junxian Guo, Wentao Ni, Yangjie Zhou, Yue Guan, Cong Guo, Weihao Cui, Yu Feng, Minyi Guo, Yuhao Zhu, Minjia Zhang, Chen Jin, and Jingwen Leng. 2025. VQ-LLM: High-performance Code Generation for Vector Quantization Augmented LLM Inference. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA61900.2025.00112

[145] David Lopez-Paz and Marc'Aurelio Ranzato. 2017. Gradient Episodic Memory for Continual Learning. In *Advances in Neural Information Processing Systems*.

[146] Chiheng Lou, Sheng Qi, Chao Jin, Dapeng Nie, Haoran Yang, Yu Ding, Xuanzhe Liu, and Xin Jin. 2026. HydraServe: Minimizing Cold Start Latency for Serverless LLM Serving in Public Clouds. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*.

[147] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 285–301.

[148] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 303–320.

[149] Yu. A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42,

4 (2020), 824–836.

[150] Yancan Mao, Shuhao Zhang, and Richard Ma. 2025. Spacker: Unified State Migration for Distributed Streaming. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*.

[151] Yancan Mao, Jianjun Zhao, Shuhao Zhang, Haikun Liu, and Volker Markl. 2023. MorphStream: Adaptive Scheduling for Scalable Transactional Stream Processing on Multicores. *Proceedings of the ACM on Management of Data* 1, 1, Article 59 (2023).

[152] Avinash Kumar Maurya, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2025. MLP-Offload: Multi-Level, Multi-Path Offloading for LLM Pre-training to Break the GPU Memory Wall. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/3712285.3759864

[153] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Proceedings of the Conference on Innovative Data Systems Research*.

[154] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[155] Zizhao Mo, Jianxiong Liao, Huanle Xu, Zhi Zhou, and Chengzhong Xu. 2025. Hetis: Serving LLMs in Heterogeneous GPU Clusters with Fine-grained and Dynamic Parallelism. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 25)*.

[156] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*.

[157] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[158] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

[159] Seonjin Na, Geonhwa Jeong, Byung Hoon Ahn, Aaron Jezghani, Jeffrey Young, Christopher J. Hughes, Tushar Krishna, and Hyesoon Kim. 2025. FlexInfer: Flexible LLM Inference with CPU Computations. In *Proceedings of Machine Learning and Systems*.

[160] Muhammad A. U. Nasir, Thomas Rausch, Andreas Kallback-Ratsch, Viktor Leis, Asterios Katsifodimos, and Volker Markl. 2019. Megaphone: Latency-conscious State Migration for Distributed Streaming Dataflows. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

[161] Nandeeka Nayak, Xinrui Wu, Toluwanimi O. Odemuyiwa, Michael Pellauer, Joel S. Emer, and Christopher W. Fletcher. 2024. FuseMax: Leveraging Extended Einsums to Optimize Attention Accelerator Design. In *57th IEEE/ACM International Symposium on Microarchitecture (MICRO 57)*.

[162] Sean Nian, Jiahao Fang, Qilong Feng, Zhiyu Wu, and Fan Lai. 2026. CacheFlow: Efficient LLM Serving with 3D-Parallel KV Cache Restoration. *arXiv preprint arXiv:2604.25080* (2026).

[163] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. 2024. ExeGPT: Constraint-Aware Resource Scheduling for LLM Inference. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[164] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference*.

[165] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2011. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2011), 92–105.

[166] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. 2025. Marconi: Prefix Caching for the Era of Hybrid LLMs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[167] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. 2025. InstAttention: In-Storage Attention Offloading for Cost-Effective Long-Context LLM Inference. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA 2025)*. https://doi.org/10.1109/HPCA61900.2025.00113

[168] Xinglin Pan, Wenxiang Lin, Lin Zhang, Shaohuai Shi, Zhenheng Tang, Rui Wang, Bo Li, and Xiaowen Chu. 2025. FSMoE: A Flexible and Scalable Training System for Sparse Mixture-of-Experts Models. In *Proceedings of the 30th*

ACM International Conference on Architectural Support for Programming Languages and Operating Systems.

[169] Zaifeng Pan, Yitong Ding, Yue Guan, Zheng Wang, Zhongkai Yu, Xulong Tang, Yida Wang, and Yufei Ding. 2025. FastTree: Optimizing Attention Kernel and Runtime for Tree-Structured LLM Inference. *Proceedings of Machine Learning and Systems* 7 (2025).

[170] Junhyeok Park, Sungbin Jang, Yongho Lee, Osang Kwon, and Seokin Hong. 2025. CLAP: Leveraging Chiplet-Locality for Efficient Memory Mapping in Multi-Chip Module GPUs. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture.*

[171] Yeonhong Park, Jake Hyun, Hojoon Kim, and Jae W. Lee. 2025. DecDEC: A Systems Approach to Advancing Low-Bit LLM Quantization. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25).*

[172] Piyush Patel, Esha Choukse, Chengliang Zhang, Tim Kraska, and Ramesh K. Sitaraman. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. *arXiv preprint arXiv:2407.03243* (2024).

[173] Daniel Peng and Frank Dabek. 2010. Large-Scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation.*

[174] Li Peng, Yuda An, You Zhou, Chenxi Wang, Qiao Li, Chuanning Cheng, and Jie Zhang. 2024. ScalaCache: Scalable User-Space Page Cache Management with Software-Hardware Coordination. In *2024 USENIX Annual Technical Conference (USENIX ATC 24).*

[175] Ameya Prabhu, Philip H. S. Torr, and Puneet K. Dokania. 2020. GDumb: A Simple Approach that Questions Our Progress in Continual Learning. In *Proceedings of the European Conference on Computer Vision.*

[176] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems.*

[177] Ruoyu Qin, Weiran He, Yaoyu Wang, Zheming Li, Xinran Xu, Yongwei Wu, Weimin Zheng, and Mingxing Zhang. 2026. Prefill-as-a-Service: KVCache of Next-Generation Models Could Go Cross-Datacenter. *arXiv preprint arXiv:2604.15039* (2026).

[178] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation - A KVCache-centric Architecture for Serving LLM Chatbot. In *19th USENIX Conference on File and Storage Technologies (FAST 25).*

[179] Jiansheng Qiu, Fangzhou Yuan, Mingyu Gao, and Huanchen Zhang. 2025. HotRAP: Hot Record Retention and Promotion for LSM-trees with Tiered Storage. In *2025 USENIX Annual Technical Conference (USENIX ATC 25).*

[180] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. 2025. Accelerating Retrieval-Augmented Generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.*

[181] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.*

[182] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. 2017. iCaRL: Incremental Classifier and Representation Learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.*

[183] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2015. Exploiting HTM and RDMA for High Performance Distributed Transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.*

[184] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. 2020. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* https://doi.org/10.1109/HPCA47549.2020.00054

[185] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Greg Wayne. 2019. Experience Replay for Continual Learning. In *Advances in Neural Information Processing Systems.*

[186] Francisco Romero, Qijian Zhao, Liyan Yao, Jiale Zhang, Xi Chen, Zhen Song, Xiaodong Wang, Tim Kraska, and Eric P. Xing. 2021. INFaaS: Automated Model-less Inference Serving. In *Proceedings of the USENIX Annual Technical Conference.*

[187] Chaoyi Ruan, Yinhe Chen, Dongqi Tian, Yandong Shi, Yongji Wu, Jialin Li, and Cheng Li. 2026. Libra: Flexible Request Partitioning and Scheduling for Serving Unbalanced and Dynamic LLM Workloads. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26).*

[188] Rya Sanovar, Srikant Bharadwaj, Renee St. Amant, Victor R"uhle, and Saravan Rajmohan. 2024. LeanAttention: Hardware-Aware Scalable Attention Mechanism for the Decode-Phase of Transformers. *arXiv preprint arXiv:2405.10480* (2024).

[189] Parth Sarthi, Salman Abdullah, Aayush Tuli, Shubh Khanna, Anna Goldie, Pranav Joshi, Danqi Chen, and Matei Zaharia. 2024. RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval. *arXiv preprint arXiv:2401.18059* (2024).

[190] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[191] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Proceedings of the International Symposium on Stabilization, Safety, and Security of Distributed Systems*.

[192] Haichen Shen, Yiding Chen, Xiaodong Wang, Wencong Ke, Yuan Wang, Bilge Acun G"unsel, Jason Lee, Junsong Song, Neeraja Yadwadkar, Ishai Menache, Daniel Mossé, Mor Harchol-Balter, Nikhil Agarwal, Tim Kraska, and Eric P. Xing. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

[193] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. 2023. S-LoRA: Serving Thousands of Concurrent LoRA Adapters. *arXiv preprint arXiv:2311.03285* (2023).

[194] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. 2024. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.

[195] Ying Sheng, Lianmin Zheng, Binhang Zhong, Siyuan Zhuang, Varun Mathew, Siheng Song, Danyang Zhuo, Ion Stoica, Joseph E. Gonzalez, and Hao Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. *Proceedings of the International Conference on Machine Learning* (2023).

[196] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *arXiv preprint arXiv:2105.09613* (2021). https://doi.org/10.48550/arXiv.2105.09613

[197] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*.

[198] Qidong Su, Wei Hao, Xin Li, Muralidhar Andoorveedu, Chenhao Jiang, Zhanda Zhu, Kevin Song, Christina Giannoula, and Gennady Pekhimenko. 2025. Seesaw: High-throughput LLM Inference via Model Re-sharding. In *Proceedings of Machine Learning and Systems*.

[199] Biao Sun, Zichen Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.

[200] Zhenbo Sun, Shengqi Chen, Yuanwei Wang, Jian Sha, Guanyu Feng, and Wenguang Chen. 2025. MEPipe: Democratizing LLM Training with Memory-Efficient Slice-Level Pipeline Scheduling on Cost-Effective Accelerators. In *Proceedings of the Twentieth European Conference on Computer Systems*.

[201] Bijan Tabatabai, James Sorenson, and Michael M. Swift. 2024. FBMM: Making Memory Management Extensible With Filesystems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*.

[202] Sisui Tang, Bingsheng He, Shuhao Zhang, and Zhenyu Niu. 2016. Elastic Multi-resource Fairness: Balancing Fairness and Efficiency in Coupled CPU/GPU Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[203] Xilin Tang, Feng Zhang, Shuhao Zhang, Yani Liu, Bingsheng He, and Xiaoyong Du. 2025. Enabling Adaptive Sampling for Intra-Window Join: Simultaneously Optimizing Quantity and Quality. *Proceedings of the ACM on Management of Data* (2025).

[204] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[205] Chunlin Tian, Xinpeng Qin, Kahou Tam, Li Li, Zijian Wang, Yuanzhe Zhao, Minglei Zhang, and Chengzhong Xu. 2025. CLONE: Customizing LLMs for Efficient Latency-Aware Inference at the Edge. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[206] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency Is the Key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*.

[207] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*.

[208] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. 2025. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association. https://www.usenix.org/conference/nsdi25/presentation/wan-borui

[209] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[210] Jiali Wang, Yankui Wang, Mingcong Han, and Rong Chen. 2025. Colocating ML Inference and Training with Fast GPU Memory Handover (SIRIUS). In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[211] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627. https://doi.org/10.1145/3448016.345755

[212] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[213] Mingqi Wang, Jun Liu, Ruicheng Zhang, Jianjun Zhao, Ruipeng Wan, Xinyan Lei, Shuhao Zhang, Bolong Zheng, Haikun Liu, Xiaofei Liao, and Hai Jin. 2026. CANDOR-Bench: Benchmarking In-Memory Continuous ANNS under Dynamic Open-World Streams. *Proceedings of the ACM on Management of Data* (2026).

[214] Qian Wang, Shuhao Zhang, and Bingsheng He. 2018. ApproxJoin: Approximate Distributed Joins for Data Analytics. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2018).

[215] Xiaoyang Wang, Yongkun Li, Kan Wu, Wenzhe Zhu, Yuqi Li, and Yinlong Xu. 2025. FineMem: Breaking the Allocation Overhead vs. Memory Waste Dilemma in Fine-Grained Disaggregated Memory Management. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[216] Xin Wang, Zhengru Wang, Zhenyu Wu, Shuhao Zhang, Xuanhua Shi, and Li Lu. 2023. Data Stream Clustering: An In-depth Empirical Study. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2023).

[217] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, Yuchen Hao, and Yufei Ding. 2025. WLB-LLM: Workload-Balanced 4D Parallelism for Large Language Model Training. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[218] Zirui Wang, Yubo Chen, Ruijie Liu, Dawn Song, and Ziyu Yao. 2024. HippoRAG: Neurobiologically Inspired Long-Term Memory for Large Language Models. *arXiv preprint arXiv:2405.14831* (2024).

[219] Zhengru Wang, Xin Wang, and Shuhao Zhang. 2024. MOStream: A Modular and Self-Optimizing Data Stream Clustering Algorithm. In *Proceedings of the IEEE International Conference on Data Mining*.

[220] Xingda Wei, Zhuobin Huang, Tianle Sun, Yingyi Hao, Rong Chen, Mingcong Han, Jinyu Gu, and Haibo Chen. 2025. PhoenixOS: Concurrent OS-level GPU Checkpoint and Restore with Validated Speculation. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

[221] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*.

[222] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. 2026. FastServe: Iteration-Level Preemptive Scheduling for Large Language Model Inference. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*.

[223] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. dLoRA: Dynamically Orchestrating Requests and Adapters for LoRA LLM Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.

[224] Yuhao Wu, Karthick Sharma, Chun Wei Seah, and Shuhao Zhang. 2023. SentiStream: A Co-Training Framework for Adaptive Online Sentiment Analysis in Evolving Data Streams. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

[225] Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song. 2024. Quant-LLM: Accelerating the Serving of Large Language Models via FP6-Centric Algorithm-System Co-Design on Modern GPUs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*.

[226] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. 2025. Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. https://doi.org/10.1145/3731569.3764815

[227] Yi Xiong, Hao Wu, Changxu Shao, Ziqing Wang, Rui Zhang, Yuhong Guo, Junping Zhao, Ke Zhang, and Zhenxuan Pan. 2024. LayerKV: Optimizing Large Language Model Serving with Layer-wise KV Cache Management. *arXiv preprint arXiv:2410.00428* (2024).

[228] Haike Xu, Magdalen Dobson Manohar, Philip A. Bernstein, Badrish Chandramouli, Richard Wen, and Harsha Vardhan Simhadri. 2025. In-Place Updates of a Graph Index for Streaming Approximate Nearest Neighbor Search. *arXiv:2502.13826* (2025). https://doi.org/10.48550/arXiv.2502.13826

[229] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In

Proceedings of the 29th ACM Symposium on Operating Systems Principles. 545–561. https://doi.org/10.1145/3600006.3613166

[230] Yanchao Xu, Dongxiang Zhang, Shuhao Zhang, Sai Wu, Zexu Feng, and Gang Chen. 2024. Predictive and Near-Optimal Sampling for View Materialization in Video Databases. *Proceedings of the ACM on Management of Data* 2, 1 (2024).

[231] Shang Yang, Junxian Guo, Haotian Tang, Qinghao Hu, Guangxuan Xiao, Jiaming Tang, Yujun Lin, Zhijian Liu, Yao Lu, and Song Han. 2025. LServe: Efficient Long-sequence LLM Serving with Unified Sparse Attention. In *Proceedings of Machine Learning and Systems*.

[232] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the European Conference on Computer Systems*.

[233] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[234] Gyeong-In Yu, Joo Seong Jeong, Gyubok Kim, Soojeong Shin, and Byung-Gon Lee. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2022).

[235] Lingfan Yu, Jinkun Lin, and Jinyang Li. 2025. Stateful Large Language Model Serving with Pensieve. In *Proceedings of the European Conference on Computer Systems*.

[236] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, Ion Stoica, Harry Xu, and Ying Sheng. 2025. Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving. *arXiv preprint arXiv:2505.04021* (2025).

[237] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[238] Yifan Yu, Yu Gan, Nikhil Sarda, Lillian Tsai, Jiaming Shen, Yanqi Zhou, Arvind Krishnamurthy, Fan Lai, Hank Levy, and David Culler. 2025. IC-Cache: Efficient Large Language Model Serving via In-context Caching. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. https://doi.org/10.1145/3731569.3764829

[239] Ziyang Yue, Bolong Zheng, Ling Xu, Kanru Xu, Shuhao Zhang, Yajuan Du, Yunjun Gao, Xiaofang Zhou, and Christian S. Jensen. 2025. Select Edges Wisely: Monotonic Path Aware Graph Layout Optimization for Disk-Based ANN Search. *Proceedings of the ACM on Management of Data* (2025).

[240] Sungmin Yun, Kwanhee Kyung, Juhwan Cho, Jaewan Choi, Jongmin Kim, Byeongho Kim, Sukhan Lee, Kyomin Sohn, and Jung Ho Ahn. 2024. Duplex: A Device for Large Language Models with Mixture of Experts, Grouped Query Attention, and Continuous Batching. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO61859.2024.00105

[241] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 423–438.

[242] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. In *Proceedings of the VLDB Endowment*, Vol. 10. 685–696.

[243] Xianzhi Zeng, Wenchao Jiang, and Shuhao Zhang. 2024. LibAMM: Empirical Insights into Approximate Computing for Accelerating Matrix Multiplication. In *Advances in Neural Information Processing Systems*.

[244] Xianzhi Zeng and Shuhao Zhang. 2023. Parallelizing Stream Compression for IoT Applications on Asymmetric Multicores. In *Proceedings of the IEEE International Conference on Data Engineering*.

[245] Xianzhi Zeng and Shuhao Zhang. 2024. CStream: Parallel Data Stream Compression on Multicore Edge Devices. *IEEE Transactions on Knowledge and Data Engineering* 36, 11 (2024), 5889–5904.

[246] Xianzhi Zeng, Shuhao Zhang, Hongbin Zhong, Hao Zhang, Mian Lu, Zhao Zheng, and Yuqiang Chen. 2024. PECJ: Stream Window Join on Disorder Data Streams with Proactive Error Compensation. *Proceedings of the ACM on Management of Data* 2, 1 (2024).

[247] Shichen Zhan, Li Li, and Chengzhong Xu. 2025. AssyLLM: Efficient Federated Fine-tuning of LLMs via Assembling Pre-trained Blocks. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 1677–1691. https://www.usenix.org/conference/atc25/presentation/zhan

[248] Chen Zhang, Kuntai Du, Siu Liu, Woosuk Kwon, Xiangxi Mo, Yufeng Wang, Xiaoxuan Liu, Kaichao You, Zhuohan Li, Mingsheng Long, Jidong Zhai, Joseph E. Gonzalez, and Ion Stoica. 2025. Jenga: Effective Memory Management for Serving LLM with Heterogeneity. *arXiv preprint arXiv:2503.18292* (2025).

[249] Feng Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2020. FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures. In *Proceedings of the USENIX Annual*

Technical Conference.

[250] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2016. Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (2016).

[251] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*.

[252] Hao Zhang, Xianzhi Zeng, Shuhao Zhang, Xinyi Liu, and Mian Lu. 2023. Scalable Online Interval Join on Modern Multicore Processors in OpenMLDB. In *Proceedings of the IEEE International Conference on Data Engineering*.

[253] Haoyang Zhang, Yirui Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. 2023. G10: Enabling An Efficient Unified GPU Memory and Storage Architecture with Smart Tensor Migrations. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 56)*.

[254] Qihao Zhang, Mingshu Zhai, Rui Sun, and Jidong Zhai. 2025. QFactory: Accelerating Quantized Large Language Model Serving with Qtile Graphs. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[255] Ruicheng Zhang, Xinyi Li, Tianyi Xu, Shuhao Zhang, Xiaofei Liao, and Hai Jin. 2026. Neuromem: A Granular Decomposition of the Streaming Lifecycle in External Memory for LLMs. In *Proceedings of the International Conference on Machine Learning*.

[256] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *Proceedings of the IEEE International Conference on Data Engineering*.

[257] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[258] Shuhao Zhang, Yancan Mao, Jiong He, Philipp M. Grulich, Steffen Zeuch, Bingsheng He, Richard T. B. Ma, and Volker Markl. 2021. Parallelizing Intra-Window Join on Multicores: An Experimental Study. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2021).

[259] Senlei Zhang, Tongjun Shi, Dandan Song, Luan Zhang, Shuhao Zhang, Xiaofei Liao, and Hai Jin. 2026. FlowRAG: Continual Learning for Dynamic Retriever in Retrieval-Augmented Generation. In *Proceedings of the ACM Web Conference*.

[260] Shuhao Zhang, Juan Soto, and Volker Markl. 2024. A Survey on Transactional Stream Processing. *The VLDB Journal* 33, 2 (2024), 451–479.

[261] Shuhao Zhang, H. T. Vo, Daniel Dahlmeier, and Bingsheng He. 2017. Multi-Query Optimization for Complex Event Processing in SAP ESP. In *Proceedings of the IEEE International Conference on Data Engineering*.

[262] Shuhao Zhang, Yingjun Wu, Feng Zhang, and Bingsheng He. 2020. Towards Concurrent Stateful Stream Processing on Multicore Processors. In *Proceedings of the IEEE International Conference on Data Engineering*.

[263] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, and Paul Johns. 2020. Hardware-Conscious Stream Processing: A Survey. *SIGMOD Record* 48, 4 (2020), 27–34.

[264] Wei Zhang, Zhiyu Wu, Yi Mu, Rui Ning, Banruo Liu, Nikhil Sarda, Myungjin Lee, and Fan Lai. 2026. JITServe: SLO-aware LLM Serving with Imprecise Request Information. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*.

[265] Yu Zhang, Feng Zhang, Hourun Li, Shuhao Zhang, and Xiaoyong Du. 2023. CompressStreamDB: Fine-Grained Adaptive Stream Processing without Decompression. In *Proceedings of the IEEE International Conference on Data Engineering*.

[266] Yu Zhang, Feng Zhang, Hourun Li, Shuhao Zhang, Xiaoguang Guo, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2025. Data-Aware Adaptive Compression for Stream Processing. *IEEE Transactions on Knowledge and Data Engineering* (2025).

[267] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*.

[268] Zhenyu Zhang, Shiwei Liu, Runjin Chen, Bhavya Kailkhura, Beidi Chen, and Zhangyang Wang. 2024. Q-Hitter: A Better Token Oracle for Efficient LLM Inference via Sparse-Quantized KV Cache. In *Proceedings of Machine Learning and Systems*.

[269] Jianjun Zhao, Wei Chen, Minchen Yu, Lei Guo, Zhipeng Li, Binyang Li, Qianxi Zhang, Yizhou Shan, and Wei Wang. 2025. Towards High-Performance Transactional Stateful Serverless Workflows with Affinity-Aware Leasing. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[270] Jianjun Zhao, Haikun Liu, Shuhao Zhang, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2024. Fast Parallel Recovery for Transactional Stream Processing on Multicores. In *Proceedings of the IEEE International Conference on Data Engineering*.

[271] Jianjun Zhao, Yancan Mao, Zhonghao Yang, Haikun Liu, and Shuhao Zhang. 2025. Scalable Transactional Stream Processing on Multicore Processors. *IEEE Transactions on Knowledge and Data Engineering* 37, 7 (2025), 4254–4269.

[272] Kaiyang Zhao, Neha Gholkar, Hasan Maruf, Abhishek Dhanotia, Johannes Weiner, Gregory Price, Ning Sun, Bhavya Dwivedi, Stuart Clark, and Dimitrios Skarlatos. 2025. Equilibria: Fair Multi-Tenant CXL Memory Tiering at Scale. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[273] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2024. Atom: Low-Bit Quantization for Efficient and Accurate LLM Serving. In *Proceedings of Machine Learning and Systems*.

[274] Youpeng Zhao and Jun Wang. 2024. ALISE: Accelerating Large Language Model Serving with Speculative Scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*.

[275] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, and Ying Sheng. 2023. SGLang: Efficient Execution of Structured Language Model Programs. *arXiv preprint arXiv:2312.07104* (2023).

[276] Size Zheng, Renze Chen, Meng Li, Zihao Ye, Luis Ceze, and Yun Liang. 2024. vMCU: Coordinated Memory Management and Kernel Optimization for DNN Inference on MCUs. *arXiv preprint* (2024).

[277] Wenxin Zheng, Bin Xu, Jinyu Gu, and Haibo Chen. 2025. SAVE: Software-Implemented Fault Tolerance for Model Inference against GPU Memory Bit Flips. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*.

[278] Binhang Zhong, Lianmin Zheng, Ying Sheng, Zhuohan Li, Hao Zhang, and Ion Stoica. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-Optimized Large Language Model Serving. *arXiv preprint arXiv:2401.09670* (2024).

[279] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.

[280] Yijie Zhong, Minqiang Zhou, Zhirong Shen, and Jiwu Shu. 2024. UniMem: Redesigning Disaggregated Memory within A Unified Local-Remote Memory Hierarchy. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*.

[281] Yuhao Zhou, Yuxin Tian, Jindi Lv, Mingjia Shi, Yuanxi Li, Qing Ye, Shuhao Zhang, and Jiancheng Lv. 2025. Ferret: An Efficient Online Continual Learning Framework under Varying Memory Constraints. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

[282] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. 2024. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO61859.2024.00111

[283] Qianchao Zhu, Jiangfei Duan, Chang Chen, Siran Liu, Guanyu Feng, Xin Lv, Chuanfu Xiao, Dahua Lin, and Chao Yang. 2024. SampleAttention: Near-Lossless Acceleration of Long Context LLM Inference with Adaptive Structured Sparse Attention. In *Advances in Neural Information Processing Systems*.

[284] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.