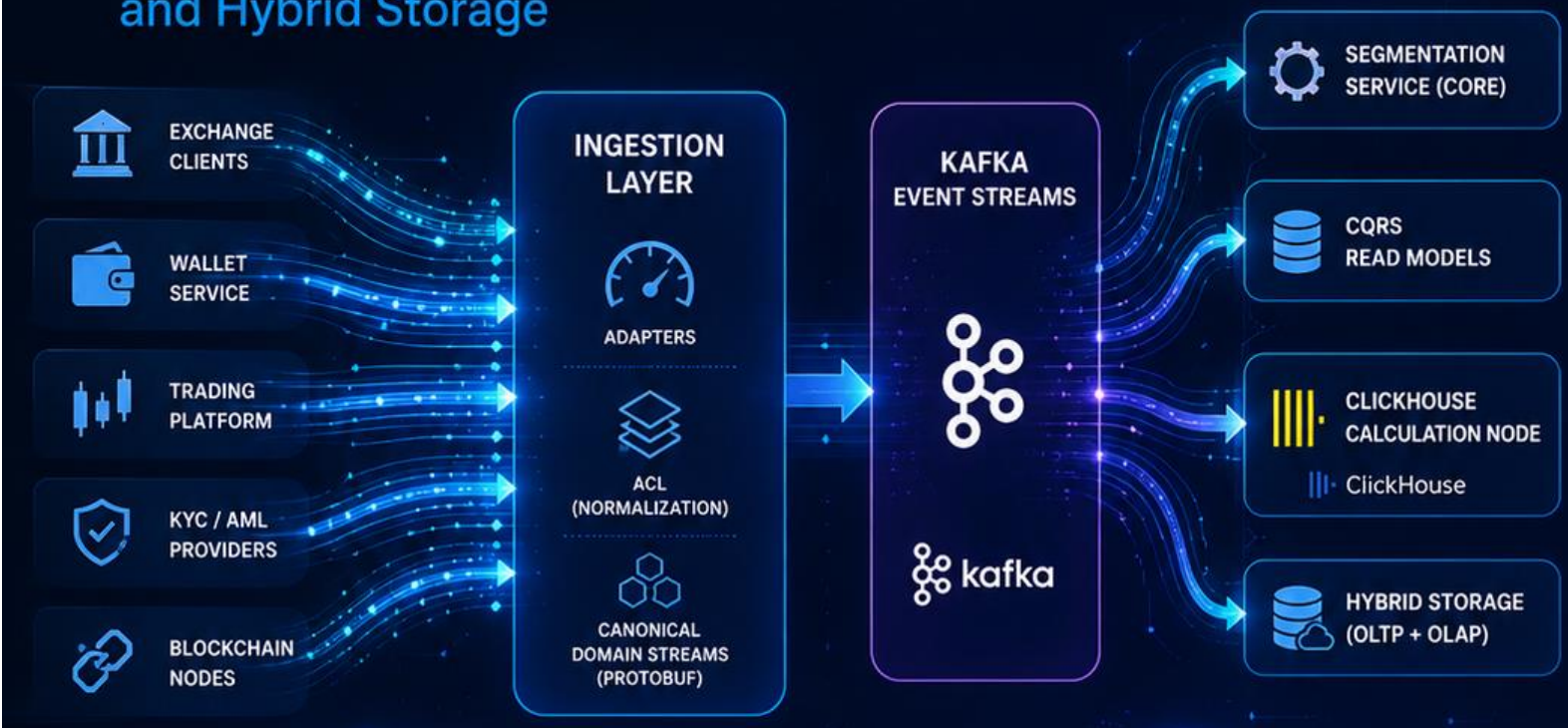
2026

Methodology for Designing

EVENT-DRIVEN HIGH-THROUGHPUT SYSTEMS

IN THE .NET ENVIRONMENT:

Ingestion Pipeline Protocol and Hybrid Storage



Serhii Yakhin

The American Journal of Engineering and Technology

Methodology for Designing Event-Driven High-Throughput Systems in the .NET Environment:

Ingestion Pipeline Protocol and Hybrid Storage

Serhii Yakhin

Senior .NET Software Engineer at Growe Hungary, Budapest

PUBLICATION INFO: The American Journal of Engineering and Technology (ISSN: 2689-0984)

ISBN: -978-1-957653-69-3

CROSSREF DOI: -https://doi.org/10.37547/tajet/mth-26-01

PUBLISHED DATE: -15 June 2026

2026

Abstract

The methodology examines a design approach for event-driven High-Throughput systems in the .NET environment, aimed at resilient processing of heterogeneous inbound traffic and suppression of degradation under stochastic load bursts. The relevance of the work stems from the limitations of synchronous REST-to-DB patterns and direct transactional writes to OLTP DBMSs, particularly in the context of the growth of Velocity–Variety–Volume, where latencies and locks increase nonlinearly, and scaling ceases to be quasi-linear. The goal of the study is to formalize an ingestion pipeline and a hybrid storage protocol that ensures the determinism of the hot path and the predictability of SLO. The novelty of the methodology lies in combining Canonical Domain Streams (ACL and Protobuf) and the Fan-In principle with asynchronous fire-and-forget ingestion on bounded channels (backpressure), as well as in reinterpreting ClickHouse as an active Calculation Node with a Stream-Back loop and a Unified Source of Truth, complemented by CQRS, hybrid caching, and an optimistic transactional outbox. Key conclusions: unification of inbound streams and decoupling of intake/processing increases resilience to spikes; migrating incremental aggregates into ClickHouse reduces load on the transactional contour; No-Magic and zero-allocation practices improve resource density and tail latencies. The methodology will be helpful to architects and engineers designing high-load streaming systems on .NET and Kafka.

Keywords: High-Throughput, .NET, event-driven architecture, ingestion, Canonical Domain Streams, Protobuf, Fan-In, System.Threading.Channels

TABLE OF CONTENTS

Introduction

In the contemporary landscape of software engineering, a fundamental shift in the design paradigm of high-load systems is observed. Traditional monolithic architectures and even classical microservice approaches grounded in synchronous interaction exhibit substantial constraints when confronted with modern load profiles. The evolution of High-Load systems has transitioned from servicing predictable data flows to governing dynamic ecosystems characterized by a high degree of uncertainty and stochastic surges of activity (Goyal & Bhasin, 2025). Modern load is determined not merely by the quantitative metric of requests per second (RPS), but by the system's ability to preserve adaptivity, data consistency, and operational resilience amid avalanche-like spikes (Spikes) and the continual complication of data topology (Dai et al., 2025).

The key problem addressed in this study is the limited suitability of classical architectural patterns, such as REST-to-DB and synchronous processing (Sync Processing), for handling high-intensity, heterogeneous inbound traffic (Çiftçi & Çiloğlugil, 2025). In big data terminology, this issue is described by the triad Velocity, Variety, Volume (speed, diversity, volume). When engineers attempt to apply standard design methods to data streams, they encounter nonlinear growth of latency, resource locking, and database degradation (Zhang et al., 2022). In particular, synchronous writes to transactional DBMSs become a bottleneck that prevents throughput from scaling linearly with respect to the inbound flow.

The objective of this work is to formalize an architectural protocol in the .NET environment for the adaptive processing of heterogeneous, high-intensity traffic. The running example throughout the methodology is an exchange domain in which WalletService and TradingPlatform generate events that update user risk classification and segmentation inside SegmentationService (Core). The proposed approach combines best engineering practices such as canonical domain streams, asynchronous ingestion layer, hybrid polyglot storage and optimized consistency patterns. The study conducts a performance evaluation of the leading-edge tools of the .NET ecosystem (for example Native AOT, System. Threading (e.g. Channels), as well as integration with specialized systems for analytical processing (e.g. ClickHouse).

Chapter 1. Ingestion Layer Protocol: Unification and Hot Path

Designing the data intake layer (Ingestion Layer) is a critical stage in building High-Throughput systems, as this component absorbs the initial load (Landry et al., 2024). The effectiveness of this layer determines the overall throughput of the entire architecture and its ability to withstand DDoS attacks or legitimate traffic surges.

1.1. The Canonical Domain Streams Strategy

Modern exchange-grade systems integrate heterogeneous inbound streams from payment providers and blockchain nodes, KYC/AML vendors, internal trading engines, and account management components (Putrama & Martinek, 2024). The fundamental issue is that such data is delivered in a dirty state: a variety of serialization formats and data transport protocols (like JSON, XML, and CSV) and schema that may change without notice. The direct propagation of this chaos into internal layers constitutes a significant architectural error, resulting in high coupling and fragility within the system core.

Anti-Corruption Layer and the introduction of Protobuf contracts

To resolve data heterogeneity, the methodology prescribes the adoption of the Canonical Domain Streams strategy. At the system boundary (Edge), a layer of thin adapters is deployed, performing the role of an Anti-Corruption Layer (ACL). These adapters, implemented using .NET Native AOT to minimize cold-start time and memory consumption, are responsible for immediate normalization of inbound data. As part of ACL's processing, various external schema formats are converted into a single, strict internal contract. The internal schema format is the Google Protocol Buffers (Protobuf) serialization format. In terms of performance, Protobuf is much more performant than text-based formats such as JSON when the traffic is heavy. The operational algorithm of the Anti-Corruption Layer is shown in Figure 1.

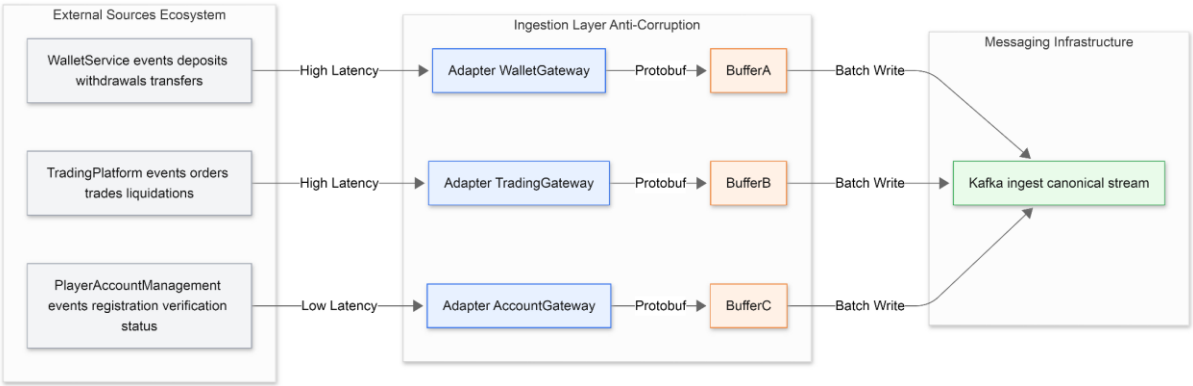


Fig. 1. Anti-Corruption Layer Algorithm

The use of binary serialization enables a reduction of the network-transmitted data volume by up to 75%, which is critically essential for decreasing load on network infrastructure and reducing transmission latency (Zandberg et al., 2024). Moreover, Protobuf serialization and deserialization require significantly less CPU time than JSON parsing, which imposes high CPU load due to string processing and allocation of transient heap objects (Viotti & Kinderkhedia, 2022). Table 1 provides a comparative analysis of serialization formats in the context of High-Load systems.

Table 1. Comparative analysis of serialization formats in the context of High-Load systems

Feature	JSON	Protocol Buffers (Protobuf)	Impact on High-Throughput Systems
Size efficiency	Low (text format, redundancy/overhead)	High (binary format, compact)	Protobuf reduces network bandwidth and disk I/O usage, enabling higher throughput of events.
CPU overhead	High (string parsing, reflection)	Low (generated code, direct byte writes)	Lower CPU usage frees resources for business logic, allowing for higher pod density.
Strong typing	Absent (schema-on-read)	Enforced (schema-on-write, `.proto`)	Protobuf prevents type errors earlier, reducing the need for defensive code in core services.
Compatibility	Complex (manual versioning management)	Built-in (forward/backward compatibility)	Simplifies system evolution without stopping services (zero-downtime deployments).

The Funnel principle (Fan-In)

A traditional integration approach often results in a situation where a central service must subscribe to dozens of distinct message-broker topics, each corresponding to an individual external source (Saleh et al., 2025). This creates an input-side Fan-Out topology that complicates scaling and renders the load profile non-deterministic. The methodology proposes inverting this approach by applying the Funnel (Fan-In) principle.

The essence of Fan-In is to unify heterogeneous inbound streams at the edge (gateways/ingestion adapters) into canonical domain streams, so that the core reads a stable and limited set of intents. Importantly, Fan-In is not a universal rule of “merge everything into one topic”. Merging unrelated domains (e.g., payments + clicks + CRM) into a single channel tends to create a “god contract”

with many optional branches, forces consumers to re-sort the traffic, and couples every consumer to every upstream schema change.

In the exchange domain used throughout this methodology, Fan-In is applied within a single bounded domain and expressed as an intent-based topology. Instead of having SegmentationService (Core) read from many source-specific topics (e.g., wallet_deposit_events, wallet_withdrawal_events, kyc_status_updates, trading_order_events), the Ingestion layer normalizes and routes events into a small number of canonical intent topics such as ingest.money-movements (deposits/withdrawals/fees), ingest.user-lifecycle (registration/verification/contact updates), and ingest.trading-activity (orders/trades). This keeps the inbound interface of SegmentationService (Core) deterministic while preserving semantic separation for consumers. Fig. 2 shows a comparison of intent-based Fan-In vs traditional source-based Fan-Out.

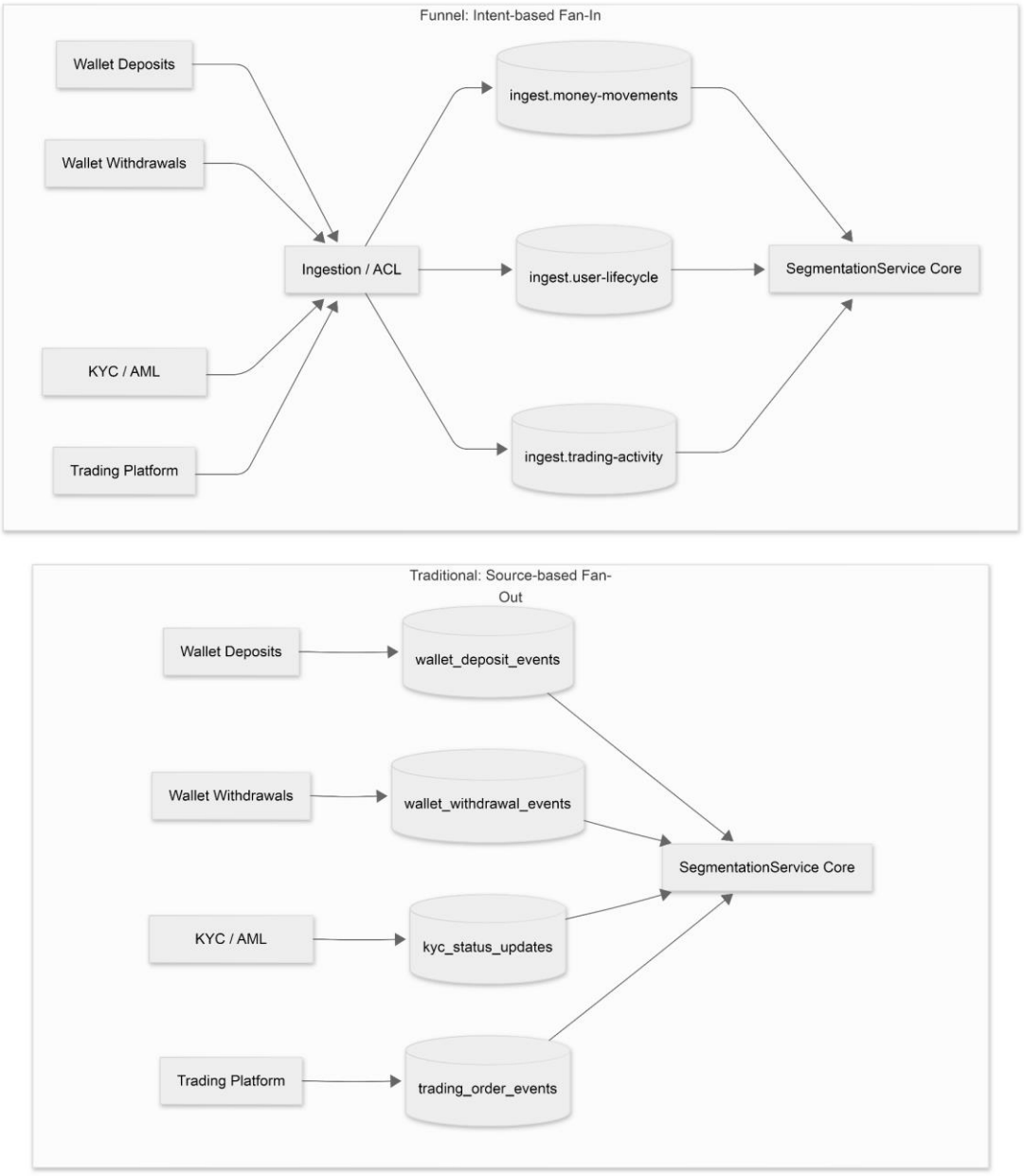


Fig. 2. Fan-In Funnel vs Traditional Fan-Out

1.2. Asynchronous Ingestion (Fire-and-Forget)

To achieve target latency metrics, the processes of request intake and request processing must be separated architecturally. However, asynchronous fire-and-forget ingestion is not a universal remedy and should be applied selectively. It is a good fit for high-volume streams that tolerate partial loss or delayed completeness (e.g., telemetry, clickstream, audit/logging, movement/observability signals) where receiving 95–99% of events is acceptable.

In contrast, business-critical exchange operations such as deposits/withdrawals, executed trades, payments, and verification status changes must not rely on a "best-effort" intake path. For these flows, the ingestion endpoint must provide a durable acceptance guarantee (e.g., a broker ack, or a persistent write as part of an outbox/ledger transaction), because losing even a small fraction of events is unacceptable and directly impacts user funds and compliance.

Dumb Producer: Minimization of ingress logic

The ingress layer of the Ingestion service should be implemented under the Dumb Producer paradigm. This implies that the ingress endpoint is entirely devoid of complex business logic and performs only two tasks: validating that the incoming message conforms to the public contract and enqueueing it into an internal buffer for asynchronous delivery to Kafka.

The concrete validation mechanism depends on the transport. If the ingress endpoint is HTTP/JSON (Minimal APIs), validation happens at deserialization time using .NET types (System.Text.Json) and optional schema validation (e.g., data annotations). The service then maps the validated DTO into the internal canonical Protobuf envelope used for the canonical domain stream. If the ingress endpoint is gRPC, the request payload is already Protobuf-based by definition, and validation is primarily contract-driven (required fields, types) plus explicit domain-level checks that cannot be expressed by the schema alone. In both cases, the goal is to keep ingress logic minimal: accept a well-formed message, normalize it into the canonical event, and hand it off to the buffering layer

In-memory buffering and spike smoothing

One of the principal challenges of High-Load systems is the non-uniformity of inbound traffic. Bursts may exceed the average load by tens of times. To smooth such micro-bursts (Burst Smoothing) before writing to Kafka, in-memory buffering is employed via System.Threading.Channels primitives.

Channels in .NET provide a high-performance, thread-safe data structure for implementing the Producer–Consumer pattern. The use of Channel<T> enables complete decoupling of the external HTTP request-reading flow from the flow that writes to the message broker. This means that a transient Kafka slowdown will not result in an immediate denial of service for inbound HTTP requests, as they will accumulate in the channel buffer. Fig. 3 illustrates Async Ingestion (Fire-and-Forget) with Dumb

Producer and In-Memory Buffer.

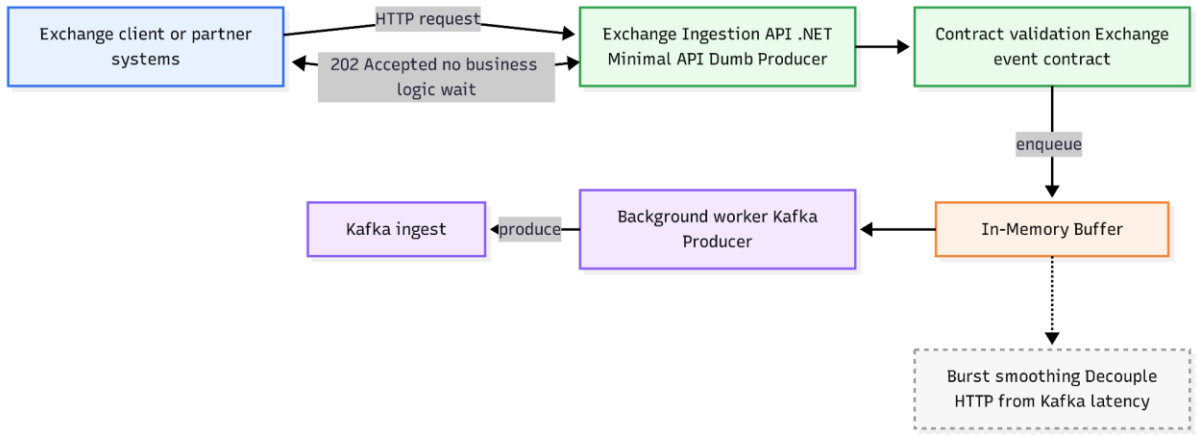


Fig. 3. Async Ingestion (Fire-and-Forget) with Dumb Producer and In-Memory Buffer

A critically important aspect is the use of bounded channels (Bounded Channels). This provides a backpressure mechanism. However, if data is arriving more quickly than it can be processed and written to Kafka for an extended period of time, the buffer will fill up: rather than allow the memory usage to continue until an OutOfMemoryException is thrown, the Ingestion service will reject requests immediately (e.g., returning a 503 Service Unavailable error). In this way, the system is predictable and will not become unusable under load.

Chapter 2. Real-Time Computation Topology: ClickHouse as a Calculation Node

In classical architectures, the database is predominantly treated as a passive state store, while the full computational burden is placed upon application servers. However, in High-Throughput systems that require analysis of event streams and real-time recomputation of user segments, such an approach proves inefficient. Transferring massive volumes of raw data from the DB into the application for computation produces enormous overhead in networking and serialization. The present methodology proposes a paradigm shift: employing an analytical DBMS, specifically ClickHouse, as an active computation node (Calculation Node).

2.1. The Computation Node pattern

Behavioral user segmentation and risk classification on an exchange often requires executing complex aggregation queries over large historical windows. For example: find all users whose net deposits exceeded 5,000 USD within the last hour, or whose withdrawal velocity exceeded a predefined threshold within a 10-minute window. However, OLTP databases, such as PostgreSQL or SQL Server, are not designed to handle such queries. Data volumes grew too large to fit into the memory that the index occupied, resulting in longer query times and locking which slowed transactional workloads.

ClickHouse, as a column-oriented OLAP DBMS, is well-suited to the role of a computational core because its architecture is oriented toward fast analytical computation over large data volumes and efficient aggregate processing.

Some engines are designed to ease streaming processing and materialized view updates using the Computation Node pattern. Kafka Table Engine allows users to stream data directly from Kafka topics into ClickHouse; likewise, the database acts as a stream consumer, allowing data to be ingested into ClickHouse with no intermediary service.

The Materialized View is used to trigger on each inserted block to invoke the next series of transformations that will populate the target structures. AggregatingMergeTree and SummingMergeTree tables are used for incrementally maintaining the state aggregates, storing the last value for every group and recomputing it every time a new block is inserted.

This pattern also encourages defining business metrics (aggregates) at Ingestion Time instead of Query Time. This enables the offloading of computational processing from .NET microservices to the database engine, which can provide SIMD CPU instruction sets and data compression to query billions of rows per second with ultra-low latencies for analytics. The Compute Node Pattern is shown in Figure 4.

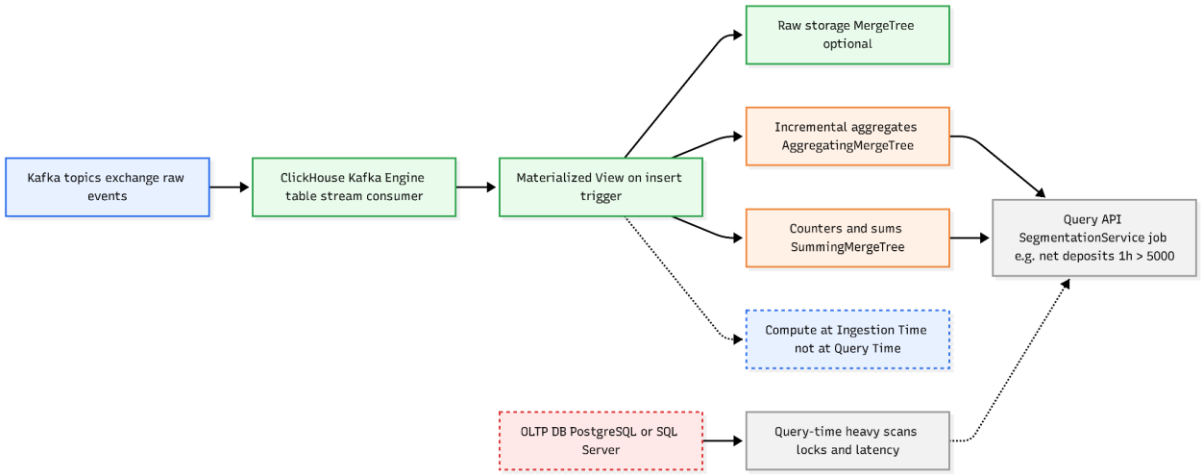


Fig. 4. Compute Node Pattern: ClickHouse as Streaming OLAP for Incremental Aggregates

2.2. Stream-Back loop

To implement a reactive architecture in which the system autonomously responds to changes in user state, a Stream-Back mechanism (feedback loop) is introduced. The results of computations executed in ClickHouse should not remain locked inside the database while awaiting polling. Instead, they should be returned to the event stream immediately.

Implementing the feedback loop implies the presence of a component (typically a RiskCalculationService) that reads aggregation results or receives notifications of threshold attainment and forms new domain events (e.g., UserBecameHighRisk, UserSegmentChanged). These events are then published back into Kafka, either into the same topic or a different one.

To fulfill this vision, SegmentationService (Core) is designed to consume two logical data streams: raw exchange user interactions (Ingestion) and computed facts (segment/risk changes produced by the Calculation Node). SegmentationService (Core) acts as the Unified Source of Truth. All streams are merged into a single channel to update user state. Figure 5 shows how multiple external and internal events can unify into a single user state.

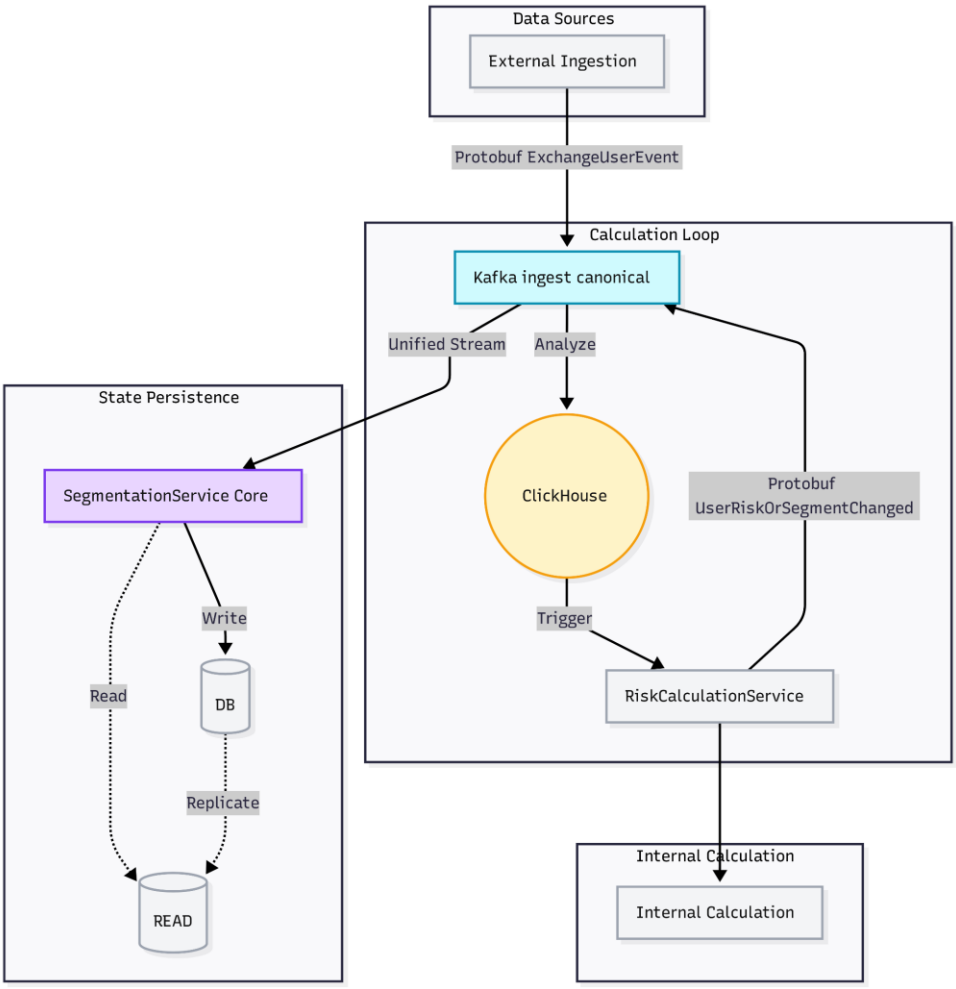


Fig. 5. Stream-Back Topology and Unified Source of Truth

This has the additional benefit that the data consumers receive these results, i.e. pre-computed Insights, rather than raw data, so the data aggregation logic need not be repeated in every data consumer, hence using the same business rules throughout the system. However, consumers of this state change only need to know that the change has occurred, not the details of the computation.

Chapter 3. State Layer: Eliminating Database Bottlenecks

Across the system, the central SegmentationService (Core) handles the high volume of events for millions of exchange users, but as the number of RPS increases past 50,000, conventional relational databases like PostgreSQL become bottlenecks (IOPS and CPU overhead) in the system. For larger workloads above 50,000 RPS (Bottleneck), using a customary database such as PostgreSQL restricts the maximum IOPS (Bottleneck) or CPU utilization (Bottleneck) for the system. Therefore a number of engineering optimizations are needed to achieve optimal performance.

3.1. Database Bottleneck Elimination

To achieve database performance, the programmer must forgo convenient abstractions, and rely instead on fine-grained control at the execution level.

First, there is the mixed data model. If all attributes are stored in separate columns (i.e., are normalized), then on frequent schema changes the ALTER TABLE statements block all access to the table. The use of the JSONB data type for flexible, frequently changing attributes, in combination with strongly typed columns for key fields (Primary Key, Foreign Key, indexes), provides a balance between performance and flexibility (Ma et al., 2025). JSONB in PostgreSQL enables the creation of GIN indexes, ensuring high-speed document search.

Second, for bulk insert operations, the Bypass ORM technique is required. ORMs like Entity Framework Core add overhead and SQL generation cost from Change Tracking. In High-Load scenarios, micro-ORMs such as Dapper or raw ADO.NET calls are preferable. PostgreSQL COPY command (with NpgsqlBinaryImporter) performs the highest because it sends data in binary format. This bypasses the SQL parser and per-row transactional overhead.

Third, aggressive batching is applied. Single-row insert operations (Single Row Insert) incur high overhead from network round-trips (Round-Trip Time) and transaction commits (WAL flush). Grouping thousands of operations into a single transaction amortizes these costs and significantly increases write throughput. Figure 6 illustrates DB Performance.

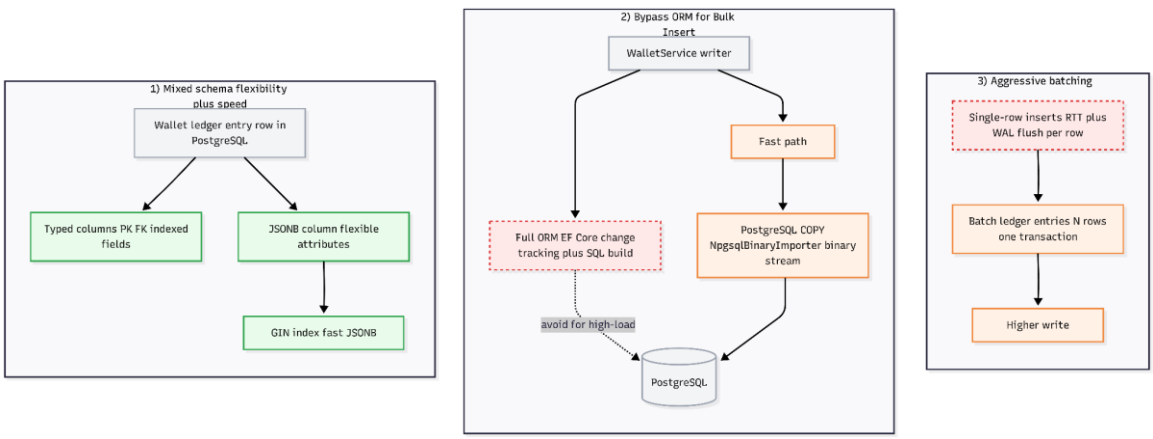


Fig. 6. DB Performance: Mixed Schema + Bypass ORM + Aggressive Batching

3.2. Partitioning Strategies

It is impractical to work with a monolithic table of tens of millions of records (70M+) since the indexes exceed the available RAM on a machine, causing Index Bloat (decreased performance) and increased disk usage.

This problem can be solved through a feature in PostgreSQL called declarative partitioning (Declarative Partitioning). Declarative partitioning is a partitioning technology that allows a table to be divided into smaller logical parts by a certain key and organized by that key.

Range Partitioning is effectively applied when data is naturally ordered by ranges, for example, for time series or for identifiers distributed across value intervals.

Hash Partitioning works in cases when an even distribution of rows across partitions is wanted and the data has no noticeable pattern like UUIDs or other random identifiers that do not correspond to range.

When the query planner partitions, it can prune them. It scans only the relevant partition when the query includes a condition on the partitioning key. This vastly speeds up query runtimes by orders of magnitude. Partitioning also simplifies the handling of data lifecycle, because obsolete data can be dropped in bulk using DETACH PARTITION, or removed by directly dropping a partitioned table rather than through an expensive and dangerous mass DELETE.

3.3. CQRS and Hybrid Caching

To improve performance further, the CQRS (Command Query Responsibility Segregation) pattern is also used, separating the read and write models of the data. The write model goes to a dedicated write database and the read model is accessed through a different path. A key read optimization is Hybrid Caching which was introduced in .NET 9 and combines the benefits of L1 local (in-memory) caching and L2 distributed (for example, Redis) caching. This allows balancing the trade-off between low latency and data coherence in a distributed environment (consistency of data across different locations).

The access to the L1 local cache happens at the nanosecond level, as only local memory accesses are involved in the lookup. The L2 distributed cache is used to keep the data coherent among the service instances, as well as to keep it persistent across pod restarts. The most important feature is the Push-based Cache Update strategy, in which instead of deleting the entry every time an entity is modified, the new value is directly pushed into the Cache through Kafka events, warming up the Cache and allowing 90% or more of the read requests to be served from the cache (Cache Hit), thus reducing the load on the database master node.

Chapter 4. Consumer Implementation: Performance and Segregation

On resource-constrained (especially cloud) infrastructure, economics means consumer code must have acceptable performance for constraints on CPU and RAM, and hence its performance must be tuned.

4.1. Single Consumer Architecture

This creates lots of small consumers and results in resource usage (each consumer reading a low-level queue has its own database connections, network sockets, and buffers). Still, it enables the Single Consumer pattern, where one BackgroundService reads a unified event stream from the low-level queues and processes it. It also allows resource coalescing, better utilization of connection pools, and less operating system context switching overhead.

4.2. Hot-Path Optimization (No-Magic Approach)

On the hot path, where code executes thousands of times per second, any hidden overhead must be eliminated. The No-Magic approach in this context refers to the refusal to use libraries that conceal internal complexity through runtime reflection, as such mechanisms introduce non-obvious execution costs and reduce performance predictability.

The Zero-Reflection concept proceeds from the premise that libraries such as MediatR (for in-process routing) and AutoMapper (for object mapping) are convenient yet costly. Reflection and dynamic dispatch introduce allocations and increase CPU load; consequently, on high-frequency code paths, cumulative latency and resource pressure increase. As an alternative, direct dependency invocations (Raw DI) and compile-time generated code (Source Generators) should be employed, yielding performance comparable to hand-written code while preserving development ergonomics.

The use of Native AOT (Ahead-of-Time compilation) in .NET 8/9 enables compiling the application directly to machine code, bypassing IL and JIT. This provides two critical advantages: minimizing cold-start time (Cold Start < 1–2 sec) and reducing memory consumption (Memory Footprint). Fast startup is essential to Kubernetes, as new pods must become operational almost instantly in response to abrupt load growth to prevent latency accumulation and lag growth.

For the effective utilization of multi-core processors within the service, internal pipelining is applied through parallel processing within the System.Threading.Channels. Data is read from Kafka by a single thread, after which it is distributed to workers through channels, and processing is performed in batches (e.g., 1000 events), which reduces synchronization overhead and increases overall throughput.

4.3. Asynchronous Consistency (Optimistic Transactional Outbox)

One of the most challenging problems in distributed systems is the dual-write problem (Dual-Write Problem), i.e., the question of how to reliably update the database and publish an event to the message broker simultaneously. The standard Transactional Outbox pattern, which involves writing to an Outbox table followed by polling-based reading, is considered a reliable approach. However, it introduces latency between committing changes and the actual publication of the event.

The proposed methodology employs an optimized variant of this approach, known as the Optimistic Transactional Outbox (Optimistic Outbox). Within a single transaction, WalletService (or TradingPlatform) writes both the business state (e.g., ledger entry or executed trade record) and the Outbox record, preserving atomicity and eliminating the risk of state divergence at commit time.

Immediately after the transaction is committed, the system, still within the application's in-memory logic, attempts to send the event to Kafka promptly and asynchronously, implementing an optimistic delivery path (Optimistic Path). If successful, the record in the Outbox is marked as sent or removed asynchronously without blocking the application, which captures the publication fact in the Outbox.

Only if it fails to send immediately (e.g., due to a broker being unavailable or a network problem) does a fallback path (Fallback Path) exist. A background poller reads unsent messages from the database and ensures their subsequent publication. As a result, this approach minimizes latency in most cases, i.e., on the happy path (Happy Path), while simultaneously preserving at-least-once delivery reliability guarantees under failures and infrastructure instability. Figure 7 illustrates an optimistic transactional outbox pattern.

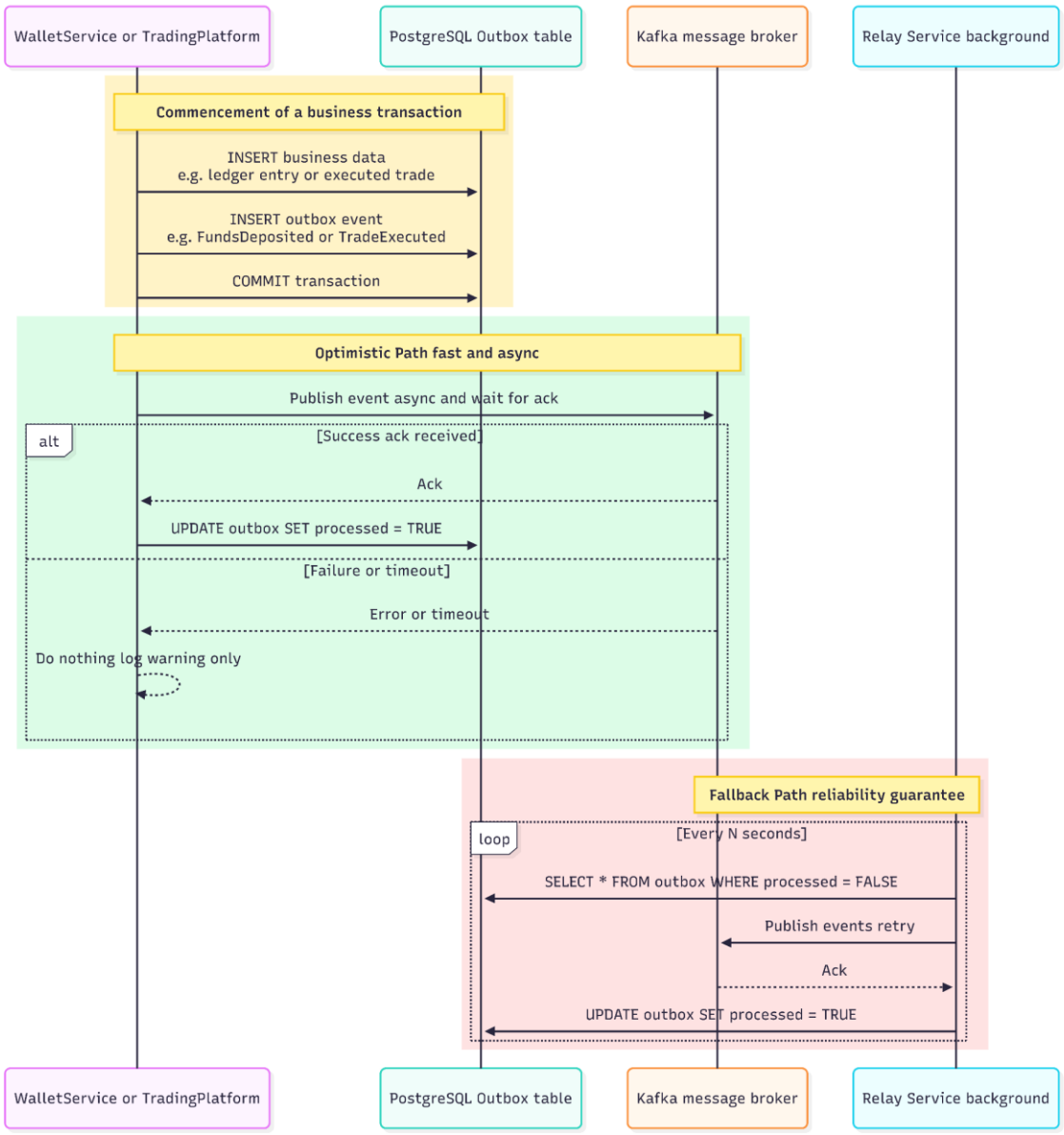


Fig. 7. Optimistic Transactional Outbox Pattern

4.4. Low-Level .NET Engineering: Zero-Allocation Techniques

To prevent Stop-the-World pauses caused by the garbage collector (GC), the volume of heap allocations must be reduced systematically, since the accumulation of transient objects and increased memory pressure raises the probability of prolonged execution halts. In practical terms, this means that on critical code sections, unnecessary memory allocations should be avoided. Whenever possible, data-processing models should be used that do not require creating new objects at each processing step.

Using Span and Memory types enables working with memory fragments and slices, without copying data. This approach is efficient for parsing strings and binary data. Instead of creating new arrays or strings, it becomes possible to operate on views of existing buffers, thereby reducing the number of transient objects managed by the GC.

The ArrayPool<T> implementation enables buffer reuse: arrays are rented from a pool and returned after the operation completes. This avoids constant allocation and deallocation of memory for new byte arrays on each I/O operation, lowering allocation frequency and reducing fragmentation, which collectively lead to more stable latencies and fewer expensive garbage collections.

Struct types are better for reducing GC pressure for small objects with short lifetimes, as they are allocated on the stack and do not require tracking by the GC. This means that the GC may potentially spend less time tracking objects, decreasing the likelihood of a pause and possibly improving predictability.

4.5. Egress Segregation (Fan-Out)

At the output of the system core, the diffusion principle (Fan-Out) is applied. The Core service should not publish all changes into one giant topic. Instead, data is segregated into thematic topics according to consumer needs (e.g., events.risk, events.wallet, events.trading, events.audit). This allows downstream services to subscribe only to data relevant to them, avoiding network and CPU overload from client-side filtering of unnecessary messages (Client-side filtering). This is referred to as Intent-Based Consumption. Egress Segregation is illustrated in Figure 8.

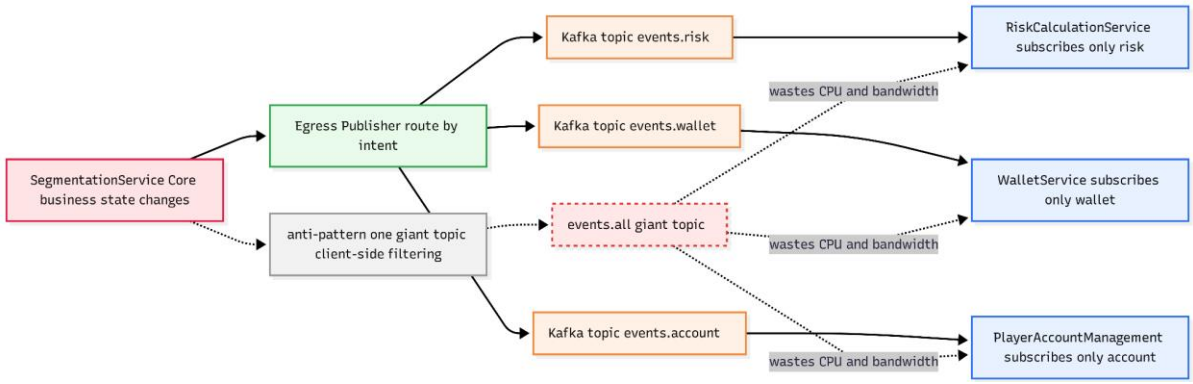


Fig. 8. Egress Segregation: Fan-Out and Intent-Based Consumption

In the diagram, the SegmentationService (Core) sends domain events to an Egress Publisher, which routes them by intent into separate Kafka topics (events.risk/wallet/trading/audit), so that each consumer subscribes only to the events it needs. An anti-pattern is shown: one giant topic events.all + client-side filtering, which leads to unnecessary CPU expenditures and network traffic for services.

Chapter 5. Reliability and Observability

Reliability in High-Load is not the absence of failures, but the system's ability to continue operating under partial component degradation.

5.1. Observability Without Degradation

Observability is critically essential for the reliable operation of high-load systems; however, monitoring tooling itself must not become a source of performance degradation. If every request is logged as structured logs under 50k+ RPS, the system begins generating gigabytes of textual data per minute. Such a stream rapidly saturates the disk and network I/O. Additionally, it consumes a significant share of the CPU for formatting, serialization, and event transmission, such that monitoring effectively begins competing with the application's valuable work.

Therefore, on the hot path, metrics should serve as the primary instrument of observability. They are substantially lighter than logs because they constitute numerical indicators that are aggregated, allowing for real-time visibility of the system state. The practical result is that rather than having to analyze every request, there can be simply treat the results as a time series of how various metrics appear to be changing, including ingest_rate, channel_depth, or db_write_latency, which is sufficient for most anomaly detection and diagnosis.

While high cardinality may be required for certain analyses, it is important to note that query cardinality directly impacts the load on the metrics storage backend; consequently, the design of light-weight monitoring remains a critical and potentially resource-intensive aspect of the architecture. In other words, high cardinality is a useful tool, but it should be enabled deliberately to avoid overloading the metrics infrastructure.

For distributed tracing (OpenTelemetry), recording 100% of traffic is also impractical, as the overhead and data volume become excessively high. Instead, sampling strategies are applied, particularly probabilistic sampling, where only a small fraction of successful requests is retained, e.g., 0.1%, to obtain a statistically representative picture. Meanwhile, error requests are captured in full, i.e., in 100% of cases, to ensure sufficient material for incident analysis. This mode preserves the informativeness of observability while maintaining monitoring cost at an acceptable level.

5.2. Resilience Patterns

Idempotency is practically required in distributed systems. Guaranteeing exactly-once delivery essentially requires the system to be stateful and violates the performance benefits of statelessness. Instead, the correct design must provide at least once semantics to cover the case where the same event may be delivered at least once and can be delivered multiple times in the worst case. In this model, the key task becomes duplicate handling, implemented at the Ingestion or Core level. Time windows and fast stores, such as Redis or Bloom Filters, are used to check the uniqueness of message identifiers, for

example, Event IDs, thereby preventing the repeated application of the same event.

However, in order to avoid data loss during deployment, particularly in the typical Rolling Update case, it cannot accept outside connections until no longer receiving SIGTERM, and has finished processing all the messages on the System. This means any channels close. Threading closes the database. Kafka connections close. The process terminates. This makes sure data in in-memory buffers is written onto persistent storage instead of being discarded when the process is signaled to terminate.

In the case during overload, like when internal channels are saturated, a load-shedding mechanism may be required. Under such conditions, it is preferable to fail fast for a portion of new requests by applying a Fast Failure strategy, thereby maintaining operability and predictable processing of already accepted messages, rather than attempting to process the entire flow and ultimately crashing due to resource exhaustion. This localizes overload damage and supports system stability in critical operating regimes. Table 2 presents high-load readiness metrics (Checklist Summary).

Table 2. High-Load Readiness Metrics

Category	Metric	Target Value	Description
Throughput	Ingestion Latency (P99)	< 50 ms	Time from request arrival to write acknowledgment (Ack) in Kafka.
Performance	Processing Lag	Consumer > Producer + 30%	Processing speed must exceed ingestion rate by at least 30% to absorb spikes.
Startup	Cold Start	< 1–2 sec	Time from pod start to readiness to accept traffic (relevant for Native AOT).
Database	DB Saturation	< 60%	CPU/IOPS utilization on the PostgreSQL master node at peak load should not exceed 60%.
Resources	GC Pressure	0 Gen2 Collections	No Generation 2 (Gen2) garbage collections on the hot processing path.

The table defines a set of SLO / non-functional requirements for a streaming system: rapid event intake (P99 ingestion latency < 50 ms to Ack in Kafka), processing with margin (the consumer is at least 30% more performant than the producer to smooth bursts), rapid recovery after release/failure (cold start 1–2 s), and avoidance of database saturation (PostgreSQL master CPU/IOPS load not exceeding 60% at peak). In aggregate, this concerns predictable latency and resilience under load: the system must not only keep up on average, but must also maintain tail latency and handle peaks without degradation.

The most stringent and potentially debatable points are GC Pressure (0 Gen2 collections on the hot path) and the master Postgres limit. Zero Gen2 implies a rigorous allocation/pooling discipline,

which may be challenging to achieve without affecting development and debugging. That tail latency matters, and stop-the-world pauses are unacceptable, is a good signal. Capping the Postgres master around 60% is an early warning sign about saturation and a risk to replication/failover. It also requires careful load decomposition (caching, batching, offloading reads to replicas, and slow query optimization), as well as continuous monitoring, to prevent it from becoming a bottleneck even when Kafka and the consumers keep up.

Conclusion

The presented methodology describes an integrated, scientifically grounded approach to designing event-driven systems within the .NET ecosystem, capable of sustaining High-Throughput loads at levels of 50k+ RPS. The strategy is based on a coherent set of architectural and engineering decisions aimed at resilient operation under high inbound-flow intensity, as well as preserving the predictability of system behavior under changes of scale.

Within this work, a conceptual transition from synchronous batch processing to a reactive streaming model is established. The introduction of the Fan-In principle and the Canonical Domain Streams strategy enables effective linearization of load on the system core, transforming stochastic external chaos into a deterministic task stream. As a result, a more controllable processing dynamics is achieved, and system behavior under scaling becomes more predictable.

The strategic role of ClickHouse as an Active Calculation Node is emphasized separately, endowing the system with hybrid computational capability. Such a choice enables the removal of up to 80% of the analytical load from transactional services by delegating heavy aggregations to a specialized engine. Consequently, the combination of .NET as an orchestration layer and ClickHouse as an analytical component effectively implements the Polyglot Persistence principle.

A substantive element of the methodology is economic efficiency, expressed through Resource Density. The application of low-level optimization techniques, including Zero-Allocation and Native AOT, as well as the refusal of excessive abstractions within the No-Magic Architecture approach, enables the attainment of extreme density of computational resource placement. This, in turn, leads to a noticeable reduction in total infrastructure cost of ownership by lowering hardware requirements in Kubernetes clusters.

Overall, this methodology provides architects and developers with a reliable foundation for constructing resilient, high-performance, and economically efficient next-generation data-processing systems.

References

Çiftçi, B., & Çiloğlugil, B. (2025). A Review of Comparative Studies on Performance Evaluation of Communication Mechanisms for Microservices. *Lecture Notes in Computer Science, 15650,* 98–113. https://doi.org/10.1007/978-3-031-96962-1_7

Dai, F., Hossain, M. A., & Wang, Y. (2025). State of the Art in Parallel and Distributed Systems: Emerging Trends and Challenges. *Electronics*, *14*(4), 677. https://doi.org/10.3390/electronics14040677

Goyal, M., & Bhasin, P. (2025). Moving from monolithic to microservices architecture for multi-agent systems. *World Journal of Advanced Engineering Technology and Sciences*, *15*(1), 2119–2124.

https://doi.org/10.30574/wjaets.2025.15.1.0480

Landry, M., Basile, E., Velempini, M., Talla, B. F., & Bomgni, A. B. (2024). The Impact of Data Ingestion Layer in an Improved Lambda Architecture. *Lecture Notes in Networks and Systems*, *824*, 325–333. https://doi.org/10.1007/978-3-031-47715-7_22

Ma, R., Zhang, K., He, Z., Jing, Y., Sean, W. X., & Chen, Z. (2025). CHASE: A Native Relational Database for Hybrid Queries on Structured and Unstructured Data. *ArXiv*. https://doi.org/10.48550/arxiv.2501.05006

Putrama, I. M., & Martinek, P. (2024). Heterogeneous data integration: Challenges and opportunities. *Data in Brief*, *56*, 110853. https://doi.org/10.1016/j.dib.2024.110853

Saleh, A., Morabito, R., Dustdar, S., Tarkoma, S., Pirttikangas, S., & Lovén, L. (2025). Towards Message Brokers for Generative AI: Survey, Challenges, and Opportunities. *ACM Computing Surveys*, *58*(1), 1–37. https://doi.org/10.1145/3742891

Viotti, J. C., & Kinderkhedia, M. (2022). A Benchmark of JSON-compatible Binary Serialization Specifications. *ArXiv*. https://doi.org/10.48550/arxiv.2201.03051

Zandberg, K., Gulati, M., Wunder, G., & Baccelli, E. (2024). Model CBOR Serialization for Federated Learning. *ArXiv*. https://doi.org/10.48550/arxiv.2401.14056

Zhang, C., Li, Y., Zhang, R., Qian, W., & Zhou, A. (2022). Scalable and quantitative contention generation for performance evaluation on OLTP databases. *Frontiers of Computer Science*, *17*(2), 172202. https://doi.org/10.1007/s11704-022-1056-2