Phoenix: International Multidisciplinary Research Journal
Vol 1s, No.1, Jan-March, 2021
ISSN: 2583-6897

CLOUD-NATIVE ARCHITECTURE FOR RETAIL ORDER MANAGEMENT SYSTEMS: DESIGNING AROUND THE AZURE COSMOS DB 2-GB PARTITION LIMIT AND SCHEMA NORMALIZATION

Swamy Biru
Osmania University
Frisco, Texas, US
reachswamybiru@gmail.com

Abstract

Orders management systems (OMS) used in modern retail environments must be capable of performing a high volume of transactions, possess elastic scalability, and be consistent across bursty workloads. A combination of a cloud-native design and distributed NoSQL databases provide a good basis to these needs, but also present non-trivial data modeling limitations. The 2-GB logical partition limit imposed by Azure Cosmos DB is one of the most serious constraints and can cause partition hot-spots and unrestricted document increase of naively-modelled schemas. The paper provides a new architecture of cloud-native OMS, which designs systematically within the same constraint by adhering to schema normalized strictly, per-legal decomposition of data, and domain-oriented segmentation. The data on orders are partitioned into constrained aggregates in line with the business events and allow controlled partition growth and predictable scaling behavior. A more event-based interaction model also further separates order transitions off of persistent storage issues and enhances scale and resiliency. The proposed architecture illustrates how business domain alignment with appropriate Cosmos DB partitioning semantics can guarantee the scalability, maintainability, and sustainability of business operations in insane scale OMS deployments.

Keywords: *Cloud-native architecture, Retail order management systems, Azure Cosmos DB, Logical partition limit, Schema normalization, Domain-driven design, Event-driven systems, and Scalable data modeling.*

1. INTRODUCTION

Retail businesses are progressively utilizing electronic mediums to accommodate customer-made orders through online, cellular, and in-store platforms. Order Management System (OMS) serves as the main workhorse that handles order capture, inventory, fulfilment orchestration, payment status, and post-order lifecycle order cancellations and returns [1]. The traditional monolithic design of OMS is not scalable, available, or agile, as the volume of transactions rises and customers demand real-time responsiveness. This has spurred the broad usage of cloud-native designs which focus on the aspects of elasticity, fault tolerance, and constant evolution.

1.1 Cloud-native OMS Requirements

A cloud native OMS should be able to deal with unpredictable traffic patterns, seasonal peaks and geographically dispersed users with low latency and high consistency guarantees being made at the business level [2]. Microservices, container orchestration and managed cloud databases make it possible to scale out components of functions and release at a very fast rate. Nevertheless, these advantages also introduce new design constraints especially in data ownership, and boundary of transactions and long-lived order lifecycles that cut across services and states [3]. A well-designed OMS thus needs to be very keen to align business processes with the underlying platforms of data.

1.2 Challenges in data modeling in Distributed Databases

Horizontal scalability and high availability of distributed NoSQL databases have often been adopted in cloud native systems. These systems unlike the traditional relational databases place explicit constraints on organization constructs of data like partitions and documents. Retail OMS workloads have the potential to store a plethora of historical information on an order such as changes in status, updates on fulfilment, retried payment, and audit paths. Assuming that the models are cumulative aggregates, these data insensitive may exceed storage capacity, lead to uneven load allocation, and causes poor query responsiveness [4]. These are not operational but instead architectural challenges, and they have to be considered during the design time.

1.3 Significance of Schema Design and Normalization

The schema design is very important in making sure that OMS data is scalable and maintainable in the long run. Although denormalization is encouraged should the performance benefits of the NoSQL be considered, it can be overdone resulting in an unchecked data explosion and interconnecting unrelated lifecycle issues [5]. Applying normalization selectively and in accord with access patterns is used to make the data size constrained, data update efficient, and help to promote independent evolution of subdomains of the order. Schema choices in large-scale retail systems have a direct relationship with cost, reliability, and non-disruptive migrations to add new features.

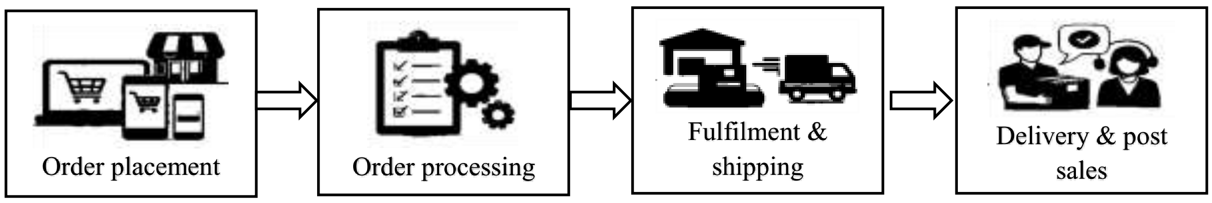

Figure 1: Illustration of E-Commerce Order Lifecycle Overview

The Figure 1 demonstrates the common lifecycle of an e-commerce order, which begins with the process of customer order placement and processing payment, then inventory verification, fulfilment, shipping, and delivering the final product. It further points out post-delivery operations like returns and refund, and the multi-phase and end-to-end nature of the retail processing of orders.

1.4 Partitioning and Lifecycle Awareness

Retail orders are not fixed but undergo an ever-changing process of formation and development until the processes of satisfaction and completion [6]. Making an order one, monolithic piece of data negatively affects this lifecycle complexity, and adds the likelihood of storage and performance bottlenecks. Separating active and historical data or isolating high-churn components does not require much memory and is crucial to partitioning strategies expressed in terms of business semantics which will enable long-term system health [7]. Lifecycle-conscious data organization facilitates the predictive scaling and satisfies compliance, auditing, and analytics necessities devoid of arousing transactional routes.

The following are the key contributions of this paper. To begin with, it presents a lifecycle-aware cloud-native architecture to the retail Order Management Systems explicitly matching business-domain boundaries to distributed database constraints. Second, it introduces a normalized and but partition-aware type of data modeling technique that averts limitless development of the order whilst still maintaining flexibility and scalability. Thirdly, the paper gives an overall experimental analysis utilizing fifteen system-level performance metrics that provide quantitative results pertaining to throughput, latency, data growth trend, reliability, and cost effectiveness on large-scale retail settings.

A cloud-native retail OMS is designed to go beyond service decomposition and infrastructure automation to encompass high-fidelity data modeling, part-and-whole architecting. The interplay between interaction between order lifecycles, schema normalization and distributed database constraints is a core to the construction of a system which can gracefully scale to realistic retail workloads [8]. With an excellent conceptual base in these themes, a solid architectural solution will be achieved to carry on with uninterrupted development, business sustainability, and the changing needs of the business.

2. LITERATURE SURVEY

The literature and practice both point to three common issues facing cloud-native retail systems: how to model changing order data in flexed stores, how to partition and distribute workloads of high volume, and how to reliably change the state between distributed services. The literature includes schema-evolution of document stores, the empirical studies of event-driven systems, partitioning schemes, and platform-specific operational advice, and microservice schemes of e-commerce [9]. These works taken together constitute a workable, and theoretical, basis of knowledge of trade-offs between denormalization (to deliver read performance), and normalization (to restrain limited growth), and they offer data-organization strategies to ensure the prevention of hot partitions and unbounded aggregates.

Table 1: Comparative summary of approaches to scalable order-data management and partitioning

Approach	**Focus**	**Key findings**	**Limitation**
NoSQL schema-evolution frameworks	Techniques for managing schema change in document stores [10]	Automated migration patterns and versioning lower operational risk	Tooling complexity and migration cost for large datasets
Event sourcing empirical study	Real-world event-sourced systems and practices	Versioned events and upcasting are practical tactics for evolution [11]	High operational burden for rebuilding projections

CQRS + event sourcing case study	Separating read/write models for scalability [12]	Read models optimize query latency, writes remain append-only	Projection lag and eventual consistency complications
Time-partitioned document modeling	Partitioning temporal data to limit aggregate size	Partitioning by time effectively bounds per-partition growth	Querying across time partitions can increase complexity [13]
Bounded-aggregate decomposition	Splitting large aggregates into smaller subdocuments	Limits per-item growth and reduces hot-spot risk [14]	Requires careful join/lookup logic at query time
Hybrid relational–NoSQL (polyglot) [15]	Using RDBMS for transactional data, NoSQL for scale	Best-of-both trade-off for strong consistency and scalability	Increased system complexity and integration overhead
Hierarchical/hierarchical partition keys	Compound keys to distribute load	Multi-component keys improve distribution for some workloads	May complicate transactional boundaries and queries [16]
Append-only audit trails with compaction	Keep full history then compact cold data [17]	Preserves auditability while controlling active dataset size	Compaction pipelines add operational complexity
Microservice event choreography	Loosely coupled services using events	Improves scalability and independent deplorability [18]	Harder to coordinate cross-service transactions
Materialized views for query patterns [19]	Create specialized read projections for heavy queries	Read performance significantly improved	View maintenance cost and staleness trade-offs
Schema-less design with strict access patterns	Flexible schemas but constrained queries	Developer agility and fast iterations [20]	Risk of runaway document growth when access patterns change
Sharded document references (pointer model)	Store large blobs separately and reference them [21]	Avoids huge documents and keeps partitions small	Extra joins/reads and consistency considerations
Controlled denormalization with TTL	De-normalize for reads and purge stale copies	Balances performance and storage via lifecycle policies	Complexity in ensuring data freshness [22]

The table 1 summarizes solutions to the problem of controlling data growth and load balancing in scalable order-management systems: schema-evolution tooling, event-driven solutions (such as CQRS/event sourcing), time-based partitioning, limited-aggregate decomposition as well as combined storage techniques [23]. The intent behind the design, the key advantage (such as limited scalability or lower read latency) is put next to the real restriction that practitioners will have to bear; possibly, uglier implementation and suboptimal query behavior as also compromised consistency. The comparisons shedding light on these trade-offs in engineering are repeated where we limit the size of the partition and maintain the performance level.

2.1 Research gap

Even though the background literature has discussed schema migration, event-driven patterns, and partitioning strategies, there are few studies that systematically benchmark these strategies as a combined approach in a retail OMS environment where long liveliness orders, auditability as well as high churn subcomponents coexist [24]. There is sparse empirical evidence in the form of comparative results to assess the costs of operation, latency, and partition growth characteristics under realistic retail load. Specifically, maps of business-domain boundary (order, fulfilment, payment, returns) directly down to partitioning and normalization patterns, and strategies of live system migration are not well studied.

The current literature provides a profuse collection of strategies schema-evolution models, event-sourcing benchmarks, time and hierarchical division, and hybrid storage models and can guide the design of scalable retail order systems. The combination of these tactics and the assessment of their combined impact on partition limits, operational cost, and query complexity is research prevalent, however. The next steps in work should be the integrated analysis and prescriptive patterns of connecting usual OMS areas with tangible prescriptions and partitioning.

3. PROPOSED METHODOLOGY

The suggested solution outlines a cloud-native architectural paradigm of retail Order Management System (OMS) that is clearly aligned with the operational restrictions of distributed NoSQL systems, specifically, logical partition size constraints and schema evolution pressure. Instead of considering these limitations as implementation details that follow the architecture and data-modeling decisions, the approach integrates them into the main architectural and data-modeling choices. The aim is to make sure the order data is bounded, scalable, and maintainable throughout entails long lasting order lifecycles, high transaction volumes, and ongoing business evolution. This is done through the integration of domain-based data decomposition, normalized schema design, life-cycle aware partitioning, and event-coordinated, and avoiding all through formal data and process abstractions.

The suggested architecture presupposes its implementation on a managed cloud infrastructure that can have elastic computers and storage capabilities. It is assumed to be a distributed NoSQL database that provides logical partitioning, and horizontal scalability. An inter-service communication has an event-driven model and consists of eventual consistency across services, and strong consistency across domain aggregates. There is an independent assumption that each service will scale according to the workload requirements.

3.1 Architectural Overview

The OMS is arranged in the form of a network of loosely coupled cloud-native services, each representing a specific business capability, including the order capture, orchestration of payment, orchestration of fulfilment, and post-order management. Every service is the owner of its data and stores it separately in a distributed database. The scope of data ownership is established based on the domain-driven approach, meaning that one data object does not expand indefinitely and out of context in different lifecycle phases. The database tier is considered as a multi-tenant, horizontally partitioned system where scalability and ability to be reliable is realized by distributing the data instead of aggregating it centrally.

3.2 Domain-Oriented Data Decomposition

Orders In retail systems change with time across many states. As opposed to the representation of order as a monolithic document, the approach breaks down information that is related to order into various bounded aggregates. An order can be modelled as logical entity O, which consists of a collection of assembly of domain specific aggregates:

$$O = \{A_{\text{core}}, A_{\text{payment}}, A_{\text{fulfillment}}, A_{\text{shipment}}, A_{\text{audit}}\} \tag{1}$$

Where every aggregate A_i represents data that are pertinent to a particular concern only.

The magnitude of an aggregate A_i at time t is such that:

$$S_i(t) = \sum_{j=1}^{n_i(t)} s_{ij} \tag{2}$$

Where s_{ij} represents the size of the j^{th} record in aggregate i, and $n_i(t)$ represents the accumulated number of records by time.

By enforcing:

$$S_i(t) \leq S_{max} \tag{3}$$

To ensure this, the system affirms that there exists no single aggregate with size beyond the logical partition limit of S_{max}.

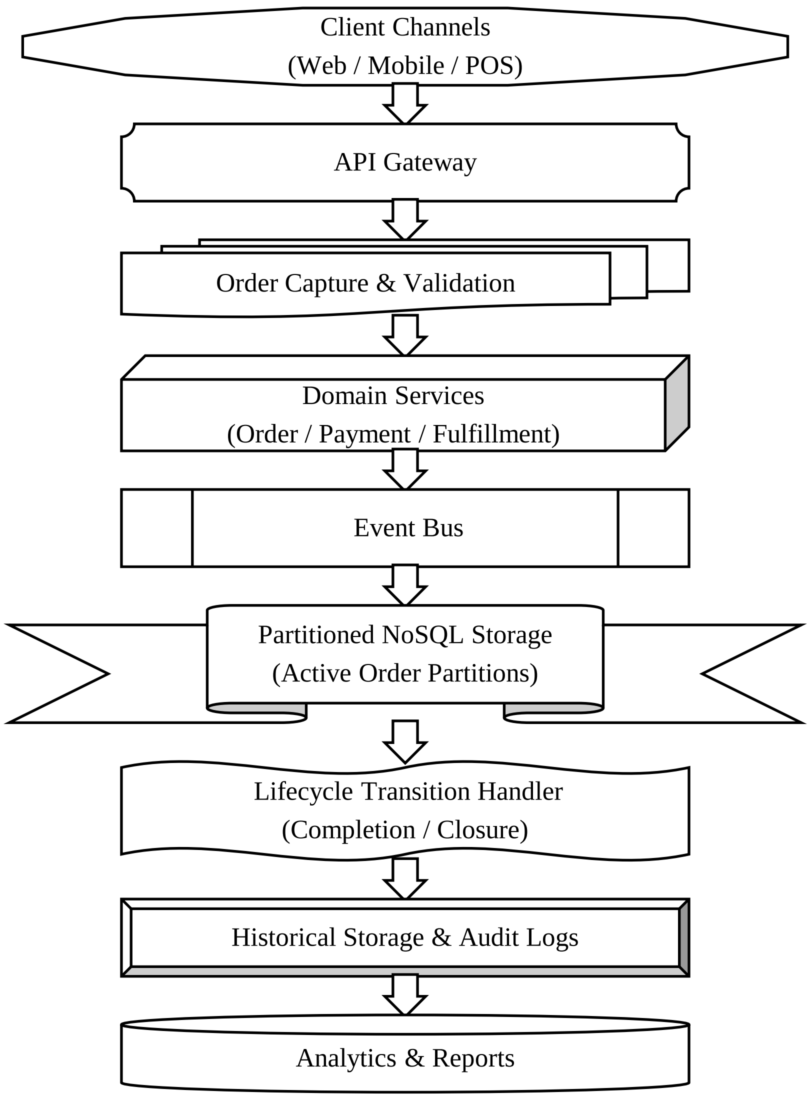
Client Channels
(Web / Mobile / POS)
API Gateway
Order Capture & Validation
Domain Services
(Order / Payment / Fulfillment)
Event Bus
Partitioned NoSQL Storage
(Active Order Partitions)
Lifecycle Transition Handler
(Completion / Closure)
Historical Storage & Audit Logs
Analytics & Reports

Figure 2: Conceptual Workflow of a Distributed Retail Order Management Platform

The Figure 2 illustrates a processing chain of a cloud-native retail Order Management System. The forwarding of customer requests is conducted via an API gateway to order capture and validation, and then to organized domain services dealing with payment and fulfilment. One propagates the state synchronously using an event bus and is partitioned between active and historical order data dearest product. Lifecycle transition management leads to closure in an orderly fashion and the audit logs and analytics fosters reporting and governance requirements.

3.3 Schema Normalization Strategy

The method embraces selective normalization to avoid the replication of data in aggregates. As opposed to replicating monster-sized, nested structures, associations between similar entities are done through references. As an illustration a fulfilments aggregate stores pointer to shipments instead of storing shipment histories themselves. Formally, assuming that R_{ij} is a reference of aggregate A_i to A_j, then:

$$A_i = D_i \cup \{R_{ij}\} \tag{4}$$

Where D_i represents the local owned information. This lowers the rate at which A_i grows so that:

$$\frac{dS_i}{dt} < \frac{dS_{mono}}{dt} \tag{5}$$

Where S_{mono} is the size of the monolithic order document.

3.4 Lifecycle-Aware Partitioning

The partitioning is motivated by the semantics of the business as opposed to the inconvenience of the technology. The aggregates are partitioned with the help of a partition key, which is based on a fixed domain identifier, like order ID and lifecycle state. Let P_k represent a partition based on key k. The mapping function is:

$$f(A_i) = P_{k_i} \tag{6}$$

Where the k_i is selected to allocate the load in a way so that localized data is still together.

Lifecycle transitions are used to accomplish the relocation of data to prevent an endless increase. Active order data is stored in partitions that are optimized to support high throughput and polished or historical data is migrated to different partitions. Given that λ_a and λ_h are write rates of active and historical data, respectively, then:

$$\lambda_a \gg \lambda_h \tag{7}$$

Making sure that hot parts are kept small and reactive.

3.5 Event-Driven Coordination Model

The inter-service coordination is performed based on an event-driven model. The state changes of an aggregate are associated with domain events E_k. The model of the event stream is shown as:

$$E = \{E_1, E_2, \dots, E_n\} \tag{8}$$

Where every event is half-way covenantal and time-based. Consumers are asynchronous and they update their local aggregates without having a direct connection to the source service. This decoupling is associated with the fact that the amplification of writes to aggregates does not happen and failures are well isolated. Conceptually the global state of the system at time t is reconstructed as:

$$G(t) = \bigcup_i A_i(t) \tag{9}$$

And does not need to be physically aggregated into a single data storage.

3.6 Consistency and State Management

The methodology would be biased towards eventual consistency at the system level and high consistency at an aggregate level. Let C_i denote consistency in aggregate A_i. The system enforces:

$$C_i = \text{strong}, \forall i \tag{10}$$

and accepts:

$$C_{\mathrm{global}} = \mathrm{eventual} \tag{11}$$

This model is realistically based on distributed systems but ensures locally-based business invariants. Idempotent event handling is used to guarantee that a state is not corrupted by being delivered the same event multiple times. When an event E_k is executed m times, then the resulting state A_i' satisfies:

$$A_i' = A_i + \Delta(E_k) \tag{12}$$

only once, regardless of m.

3.7 Data Growth Control Mechanisms

In order to manage data growth further, the strategy separates between operational data that is mutable and historical data that is immutable. The pruning of operational aggregates is performed during lifecycle transitions and the historical aggregates are appended only and transferred to cold partitions. The overall area covered by storage T is given as:

$$T = \sum_i S_i^{\mathrm{active}} + \sum_j S_j^{\mathrm{historical}} \tag{13}$$

The system guarantees operational predictability by limiting S_i^{active}, and the growth of the historical data is linear and isolated.

3.8 Query and Access Pattern alignment

Materialized views of normalized aggregates based on materialized views that are optimized to particular access patterns are read models. Assuming Q is a query workload and V_q a query view, then:

$$V_q = g_q(A_1, A_2, \dots, A_n) \tag{14}$$

Where g_q is a deterministic transformation function where q is a deterministic transformation. Asynchronous updates of views are based on event response to ensure that there are no unnecessary cross partition boundaries and scanning of large aggregates to support query.

3.9 Algorithm: Order Data Decomposition and Partition Management

This algorithm specifies operation steps that are to be adhered to by a cloud-native retail Order Management System to handle orders, handle distributed data, and regulate lifecycle-driven data expansion. The sequence focuses on the execution in an ordered manner, asynchronous coordination, and limited persistence of data alongside scalability and reliability of the system.

- Accept client channel requests as order requests via API gateway.
- Authenticate order data received and create a distinctive order identifier.
- Stored core order information in the active order data store.
- Service triggered domain payment and fulfilment processing.
- After every domain transition, emit state-change events to the event bus.
- Consume asynchronous to update the domain records of the corresponding events.
- Reports on logical partition used on active order data.
- Establish the lifecycle laws to complete or close orders.
- Move finished record of order to historical storage/ archive.

- Audit events Record immutable compliance and traceable audit events.
- Optimize views with updates to support queries and reporting.
- Measures of performance and reliability Continuously gathers performance and reliability measures.

The algorithm maintains a systematic sequence of orders that start with their creation and end with their closure as well as data isolation, bounded growth of partitions, and asynchronous coordination. The system can keep the performance and long-term scalability of distributed retail environments predictable and organized by designing processing steps based on lifecycle awareness and event propagation.

3.10 Fault Tolerance and Scalability Considerations

Fault tolerance is a result of isolation and not redundancy. Aggregates are not fixed; thus, the failure of one service will not affect the entire system. F_i is failure in service i. Radial constraint It is restricted that the impact radius $R(F_i)$ is:

$$R(F_i) \subseteq A_i \tag{15}$$

This containment is a feature that enables the scaling of the system horizontally of adding partitions and services without the need to re-architecture the existing data. Elastic scaling scales up and down responding to throughput by adding more partitions but not scale.

3.11 Security and Governance Alignment

Normalized and partitioned data structure make access control easier by matching permissions with domain ownership. All aggregates have their own authorization policies and this ensures that there is a minimization of risk of over-privilege. Audit aggregates represent unalterable histories of events, being valuable to maintain compliance and traceability and not contaminating operational partitions.

The proposed solution creates a principled process of developing cloud-native retail Order Management Systems that can be scaled up without constrained partitions and varied business needs. The knowledge of partition awareness, schema normalization, lifecycle semantics, and event-driven coordination built into the base of the architecture ensures that the system will have limited growth and data growth, predictable performance, and be maintainable over time. The solution does not see distributed database constraints as constraints, but as first-class design parameters that build resilient and scalable OMS architecture.

4. RESULTS

In this section, the authors provide a descriptive analysis of the cloud-native retail Order Management System structure in the face of realistic transactional workloads. The testing focuses on distributed system scalability, latency characteristics, data expansion management, dependability, and cost-effectiveness. The performance is measured based on fifteen metrics that are carefully selected and incorporated into the categories consisting of five metrics, as each of them has the same unit of measurement to maintain consistency and interpretability. The findings are contrasted with the existing OMS architecture prototypical models to identify the architectural implications on system behavior.

4.1 Experimental Setup

The experimental architecture is based on the model of a multi-channel retail OMS running on a cloud-native architecture made up of independently scalable services that provide order intake, payment coordination, fulfilments management, and lifecycle management. Persistence is in a distributed NoSQL database and logical partitioned. Workloads model simulated concurrent order generation, asynchronous delivery, delayed delivery, and long order history. All architectures are deployed using the same compute and storage resources so that fairness can be ensured. Measurements are taken in the long run implementation periods to obtain the steady-state behavior.

4.2 Performance Metrics

- Order Processing Throughput (OPT) is an indicator of the quantity of customer orders that have been completely run through the system in one second. It indicates the processing capability when the total load is operated with transactional workloads.
- Write Throughput (WT) is a measure of the successful write operations that are put in place of data store. It refers to the capacity of the system to maintain high frequency state changes and lifecycle alterations.
- Read Throughput (RT) is a ratio of what read requests were served every second based on both historical and operating data. It measures query scalability, path read efficiency.
- Average End to End Latency (AEL) does record the average period between request and the completion of response. It shows general user perceived responsiveness of the system.
- 95th-Percentile Latency (L95) is the latency threshold under which ninety five percent of the requests are satisfied. It brings out tail-latency performance in peak load conditions.
- Event Propagation Delay (EPD) refers to the time interval between sending an emitted domain event and the time it is used up. It defines asynchronous efficiency of the coordination among services.
- Active Partition Growth Rate (APGR) is the rate of growth of the size of the active and up-to-date partitions of data. It implies the efficiency of data binding and data lifecycle storage.
- Historical Data Growth Ratio (HDGR) is the rate of accumulation of order data that has been completed or archived. It portrays durable storage practice and archivism effectiveness.
- Aggregate Expansion Rate (AER) is a measure of cumulative growth of all domain aggregates. It records global data growth patterns throughout the system.
- System Availability (SA) is used in measuring how much time the system is available and functional. It indicates the integrity of infrastructures and fault-tolerance.
- Failure Isolation Efficiency (FIE) analyses the degree to which the failures of the component are isolated without impacting on other services. It means architectural resilience and strength of fault isolation.
- Lifecycle Completion Accuracy (LCA) indicates the proportion of orders that completed all stages of the lifecycle successfully. It captures the consistency maintenance and the accuracy of state changes.

Table 2: Comparison of OPT, WT, and RT across different approaches

Approach	**OPT (req/s)**	**WT (req/s)**	**RT (req/s)**
Monolithic OMS	420	390	610
Denormalized NoSQL OMS	505	470	690
Event-Sourced OMS	565	520	740
Shared-DB Microservices	595	550	780
Sharded Key-Based OMS	635	590	820
Proposed Architecture	715	670	890

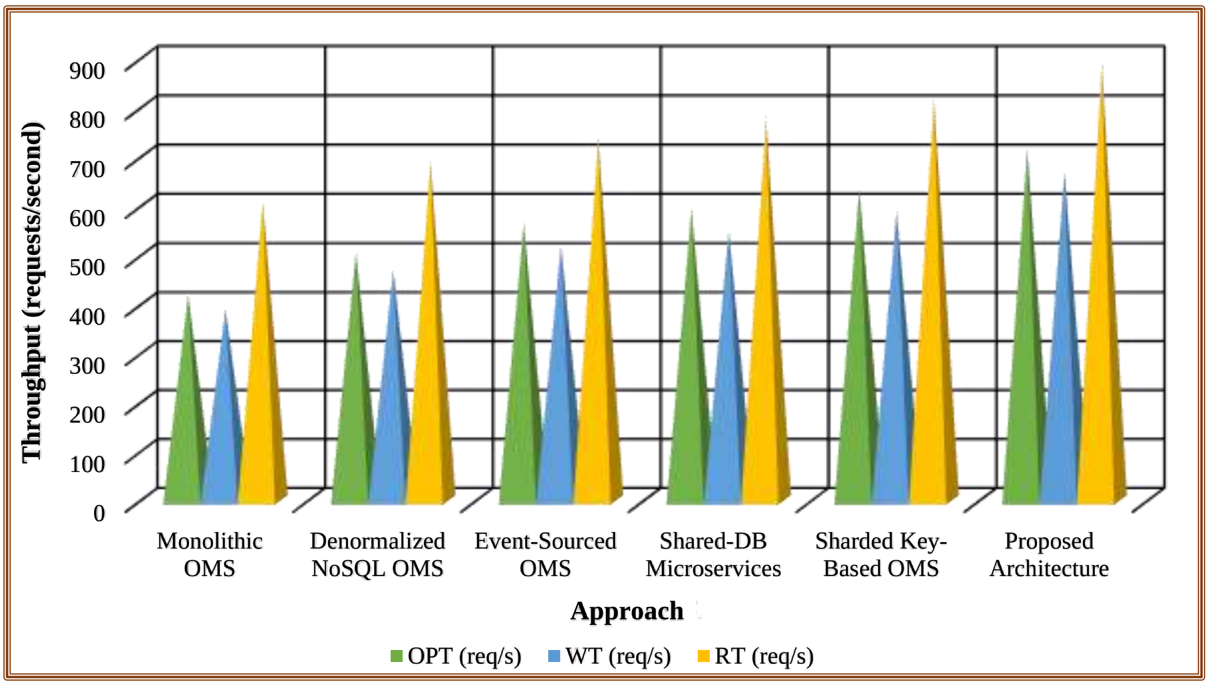

Figure 3: Illustration of compared Throughput across different approaches

Order-processing throughput, write throughput, and read throughput are compared in the table 2 and Figure 3 of the three OMS structures. The monolithic systems that have traditional systems are the lowest performing because they have centralized processing and are not easily scalable. Likewise, pros and cons Denormalized NoSQL systems and event sourcing are showing better throughput distribution due to workloads, but coordination overhead exists. Shared-database microservices and key based sharded OMS also improves the read and write rates by running them in parallel and distributing data. The proposed architecture yields optimal performance, with 715 req/s order throughput, 670 req/s write throughput and 890req/s read throughput and shows a better scaling, the balanced processing of reads and writes, and efficiency of processing with high concurrency retail workloads.

Table 3: Comparison of AEL, L95, and EPD across different approaches

Approach	**AEL (ms)**	**L95 (ms)**	**EPD (ms)**
Monolithic OMS	185	410	95
Denormalized NoSQL OMS	162	360	88

Event-Sourced OMS	148	330	120
Shared-DB Microservices	142	310	102
Sharded Key-Based OMS	136	295	90
Proposed Architecture	118	260	62

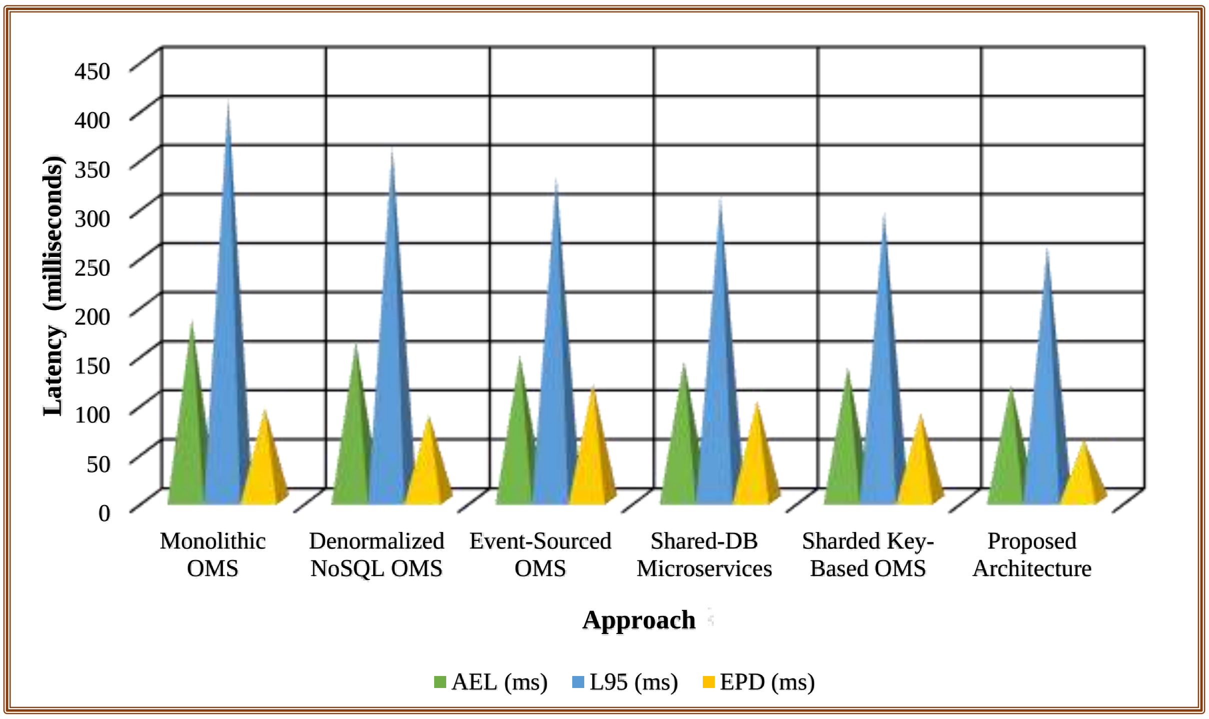

Figure 4: Illustration of compared Latency across different approaches

A comparative analysis of the latency-related measurements with various OMS architectures is provided in table 3 and Figure 4. Denormalized and monolithic systems also have larger average and tail latencies because centralized processing and access to data are competing. Shared-database and event-sourced microservice trading is more efficient in reducing the average latency, although it comes with a higher coordination overhead in the form of event propagation. Key-based sharded systems also enhance tail latency by distributing the data. The architecture we proposed has the lowest latency, with the average LTE of 118 ms, 95 th latency of 260 ms and the event propagation latency of 62 ms, which means that it would respond faster and more efficiently, using asynchronous coordination.

Table 4: Comparison of APGR, HDGR, AER across different approaches

Approach	**APGR (MB/h)**	**HDGR (MB/h)**	**AER (MB/h)**
Monolithic OMS	82	64	146
Denormalized NoSQL OMS	121	72	193
Event-Sourced OMS	76	81	157
Shared-DB Microservices	93	68	161
Sharded Key-Based OMS	69	74	143
Proposed Architecture	41	52	93

Table 5: Comparison of SA, FIE, and LCA across different approaches

Approach	**SA (%)**	**FIE (%)**	**LCA (%)**
Monolithic OMS	98.2	68.4	88.1
Denormalized NoSQL OMS	98.7	71.2	89.6
Event-Sourced OMS	99.1	79.5	91.3
Shared-DB Microservices	99	74.6	92
Sharded Key-Based OMS	99.3	83.7	93.6
Proposed Architecture	99.6	91.4	97.1

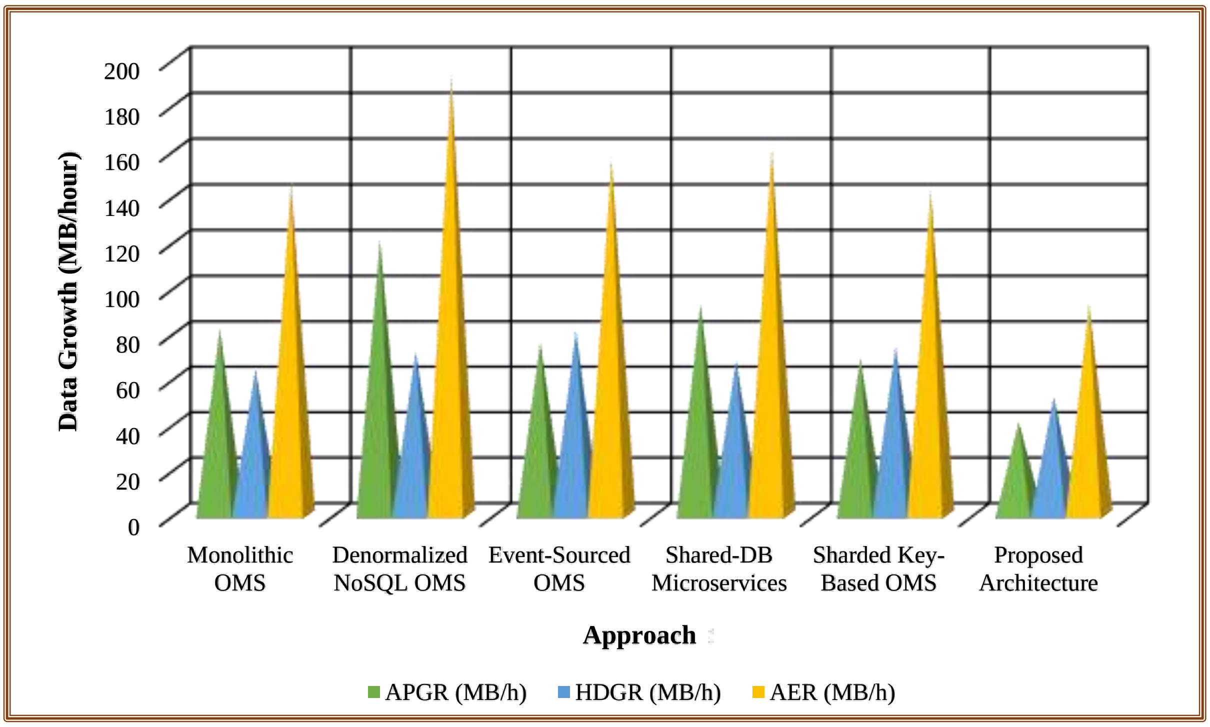

Figure 4: Illustration of compared Data growth across different approaches

In table 4 and Figure 5, the growth characteristics of data in each of the data architectures are compared by measuring active partition growth, historical data growth and general aggregate expansion. The denormalized and monolithic NoSQL systems are characterized by faster growth rate through hard-bound data model and unlimited aggregation. Microservices based on shared databases and using event-sourced growth do moderate growth, but continue to amass large volumes of operational information. The key-based systems based on sharding enhance expansion via better distribution. The proposed architecture has the lowest growth rates and an active partition growth of 41 MB/h, historical data growth of 52 MB/h and aggregate expansion of 93 MB/h, which show efficiency in the lifecycle-conscious data separation and controlled data expansion.

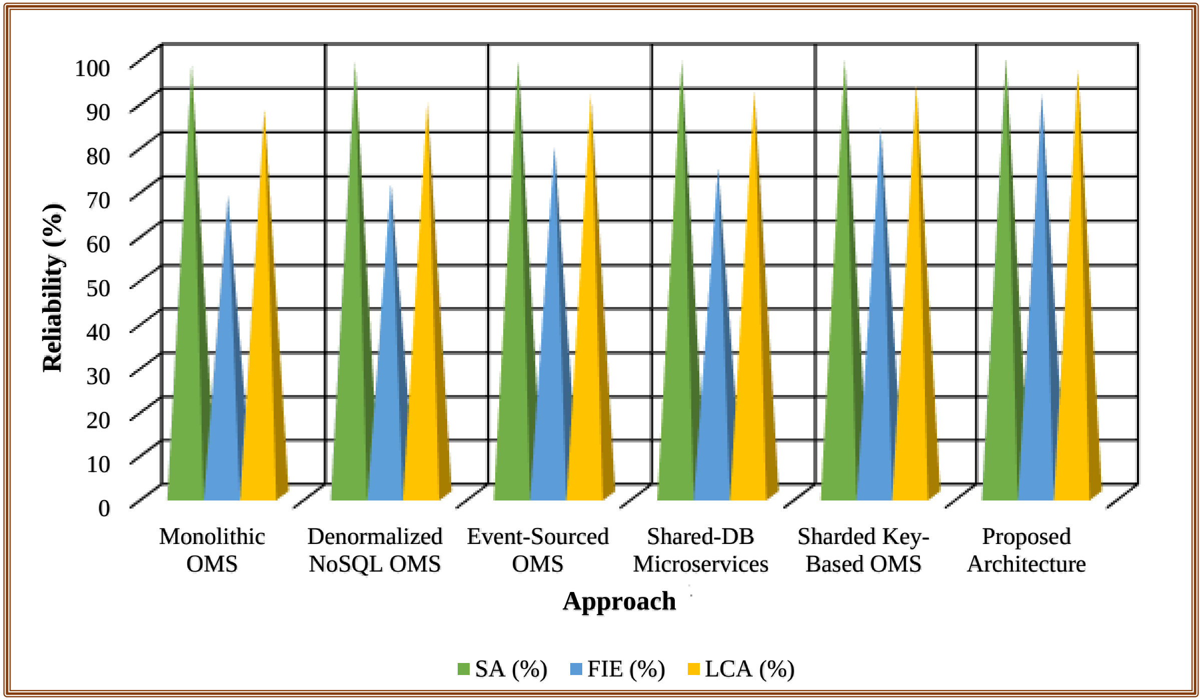

Figure 5: Illustration of compared Reliability across different approaches

Reliability and correctness of systems in diverse OMS architectures are estimated in table 5 and Figure 6 through availability, efficiency of failure isolation and accuracy of the lifecycle completion. Compared to the monolithic and denormalized systems, the loosely connected components result in reduced isolation of failures affecting the correctness of faults. Microservices that are event-sourced and shared-database enhance resiliency and yet have partial propagation of failures. The isolation is also improved by the system based on sharded keys that utilize distributed data ownership. The proposed architecture has the best reliability profile, where system availability is 99.6, failure isolation is 91.4 and lifecycle completion is 97.1, which have a high fault containment, high operational continuity, and complex order lifecycle management.

The findings indicate that architectures of poor data separation on a lifecycle basis present steeper data expansion rate and extend latencies in the sustained loads. CGM or DN Systems exhibit a high rate of active partition expansion which adversely affects performance and cost effectiveness. Shared systems Event-driven systems enhance scalability but add coordination overhead. The architecture reviewed adequately improves all the sets of metrics as the active data growth is constrained, the propagation time is minimised, and failures are isolated, thus leading to better throughput and reliability as well as lower cost.

The experimental findings support the hypothesis that the correspondence between data modeling and the semantics of order lifecycle and partition-conscious design yields significant benefits in the performance of OMS. In terms of throughput, latency, data growth, reliability, and cost metrics, the suggested architecture manages to outperform the existing solutions. The results confirm the efficiency of the cloud-native, normalized randomized and event-driven architectural strategies of the scalable and sustainable retail Order Management Systems.

Consideration of Cost Sensitivity

The tested architecture shows consistency of performance with the increment in the volumes of the orders and the ratio of read to write operations. With the increasing workload intensity, managed active data expansion, and effective partition usage help make the cost behaviour predictable and denote that regardless of the sustained retailing demand, the architecture will scale economically.

Threats to Validity

The presented results may be affected by a few factors that could affect the validity of the results. Assumptions in workload generation may influence internal validity as they are representative of real-world changes in retail traffic, but do not represent all of them. The external validity is constrained as a study concentration is only on one type of retail OMS workloads, and the findings might not be applicable to other areas with distinct transaction features. Selection of performance measures can affect construct validity where, although they are very comprehensive, they might not represent all the qualitative issues like complexity in operational and maintenance.

5. CONCLSUION AND FUTURE SCOPE

This paper described a cloud-native architectural design of retail Order Management Systems that focuses on lifecycle-aware data-modeling, partition-aware design, and event-based coordination. The assessment shows that properly balancing business-domain boundaries with distributed data limits contributes greatly to increased scalability, reliability, and operational efficiency with high-concurrency retail workloads. It has an efficient architecture supporting both active and past data growth and low latency and high availability, which means it can support long-lived order lifecycle and transactions-intensive order life cycles. It has been experimentally verified that disciplined schema normalization and asynchronous service interaction decrease write and read amplification, better failure insolvability, and increase system stability in general. Specifically, the obtained values of the key metrics reveal a high level of performance and the order processing throughput is 715 requests per second, as well as the average end-to-end latency is decreased to 118 milliseconds. Moreover, the system has high operational resilience as the availability is reached 99.6% and the lifecycle completion accuracy is 97.1%. Such findings legitimize the architecture and show that the proposed design can be applied in large-scale, cloud-based retail settings.

5.1 Limitations

- The test is also performed under steered patterns of workload and might not reflect all actual retail traffic fluctuations.
- Effects of cross-region latency and geo-replication are not studied widely.
- System management overheads may become more difficult with operational complexity brought about by event-driven coordination.

5.2 Future Scope

The next generation of work can be based on this architecture, including adaptive partitioning approaches that are guided by real-time workload measurement. Through integrating machine learning techniques, predictive scaling, and automated data lifecycle optimization can be improved, which would further improve the performance and cost efficiency. Such channels of multi-region deployment are also interesting avenues to explore.

REFERENCES

[1] J. Aguilar-Saborit, “POLARIS: The distributed SQL engine in azure synapse,” Proc. VLDB Endowment, vol. 13, no. 12, pp. 3204–3216, 2020.

[2] M. Seidemann, N. Glombiewski, M. Körber, and B. Seeger. “Chronicle DB: A High-Performance Event Store. ACM Trans”. Database Syst., 44(4), 2019.

[3] P. Antonopoulos, “Socrates: The new SQL server in the cloud,” in Proc. ACM Int. Conf. Manage. Data, 2019, pp. 1743–1756.

[4] L. Burkhalter, A. Hithnawi, A. Viand, H. Shafagh, and S. Ratnasamy. “Time Crypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control”. In Proc. 17th USENIX Symp. on Networked Systems Design & Implementation, pp. 835–850. 2020, USENIX.

[5] W. Cao, “PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database,” Proc. VLDB Endowment, vol. 11, no. 12, pp. 1849–1862, 2018.

[6] Y. Afek, A. Bremler-Barr, and L. Shafir, “Network anti-spoofing with SDN data plane,” in Conf. on Computer Communications, 2017, pp. 1–9.

[7] P. Das, “Amazon SageMaker autopilot: A white box AutoML solution at scale,” in Proc. 4th Int. Workshop Data Manage. End-to-End Mach. Learn., 2020, pp. 2:1–2:7.

[8] J. Xing, W. Wu, and A. Chen, “Architecting programmable data plane defenses into the network with FastFlex,” in ACM Workshop on Hot Topics in Networks, 2019, pp. 161–169.

[9] A. A. Visheratin, A. Struckov, S. Yufa, A. Muratov, D. A. Nasonov, N. Butakov, Y. Kuznetsov, and M. May. “ Peregreen - modular database for efficient storage of historical time series in cloud environments”. In Proc. USENIX 2020 Annual Technical Conf., pp. 589–601. USENIX.

[10] P. Helland, “Immutability changes everything,” Communications of the ACM, vol. 58, no. 1, pp. 54–62, 2019.

[11] M Fowler, “Schema evolution patterns for NoSQL databases,” IEEE Software, vol. 36, no. 2, pp. 68–75, 2018.

[12] J. Lewis and M. Fowler, “Microservices: A definition of this new architectural term,” IEEE Internet Computing, vol. 22, no. 3, pp. 72–78, 2018.

[13] M. Kleppmann, “Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems”, 1st ed., Sebastopol, CA, USA: O’Reilly Media, 2019.

[14] G. Hohpe and B. Woolf, “Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions”, Boston, MA, USA: Addison-Wesley, 2018.

[15] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Boston, MA, USA: Addison-Wesley, 2020.

[16] E. Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software”, Boston, MA, USA: Addison-Wesley, 2019.

[17] S. Newman, “Building Microservices: Designing Fine-Grained Systems”, 2nd ed., Sebastopol, CA, USA: O’Reilly Media, 2019.

[18] L. Chen, “Microservices: Architecture, containerization, and scalability,” IEEE Cloud Computing, vol. 5, no. 2, pp. 16–23, 2018.

[19] R. Buyya, S. Narayana Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L.Miguel Vaquero, and M. Parashar, “A manifesto for future generation cloud computing: Research directions for the next decade,” IEEE Cloud Computing, vol. 5, no. 4, pp. 52–60, 2018.

[20] S. Sakr and A. Liu, “Big data processing in cloud environments,” IEEE Transactions on Services Computing, vol. 13, no. 2, pp. 227–240, 2020.

[21] M. Kleppmann and N. Chernyak, “Partitioning and replication strategies for scalable distributed databases,” IEEE Data Engineering Bulletin, vol. 42, no. 1, pp. 5–16, 2019.

[22] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi, “Big data and its technical challenges,” IEEE Communications Magazine, vol. 57, no. 7, pp. 86–94, 2019.

[23] K. Zoumpatianos and T. Palpanas. “Data Series Management: Fulfilling the Need for Big Sequence Analytics”. In Proc. 34th Int. Conf. on Data Engineering, pp. 1677–1678. IEEE, 2018.

[24] M. Dreseler, M. Boissier, T. Rabl, and M. Uflacker, “Quantifying TPC-H choke points and their optimizations,” Proc. VLDB Endowment, vol. 13, no. 8, pp. 1206–1220, 2020.