iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

SERVERLESS COMPUTING PATTERNS FOR SCALABLE RETAIL TRANSACTION PROCESSING

AN ARCHITECTURAL STUDY OF EVENT-DRIVEN, FUNCTION-AS-A-SERVICE DESIGN PATTERNS FOR HIGH-VOLUME RETAIL TRANSACTION SYSTEMS

BharathVamsi Reddy Munisif
Java Developer, USA

ABSTRACT
Retail transaction processing systems face highly variable demand, with load ratios between baseline and peak promotional events frequently exceeding twenty to one. Traditional capacity planning approaches built around fixed fleets of virtual machines or containers struggle to absorb this variability without either overprovisioning for the rare peak or risking checkout failures during it. Serverless computing, in the form of Function as a Service (FaaS) platforms combined with managed event, queue, and database services, offers a computing model in which capacity scales automatically with request volume and cost tracks usage rather than provisioned capacity. This article presents a structured study of serverless architecture patterns applicable to scalable retail transaction processing, including event-driven order ingestion, choreography and orchestration based saga patterns for distributed transactions, fan-out and fan-in patterns for parallel inventory and pricing checks, command query responsibility segregation (CQRS) with serverless read replicas, circuit breaker and retry patterns for downstream resilience, and stream processing patterns for real-time analytics on transaction data. A reference architecture is proposed that combines an API gateway, function compute, managed queues and event buses, a workflow orchestration service, and a mix of key-value and relational serverless data stores. The patterns are evaluated using a benchmark methodology covering cold start latency, throughput scaling, error rates under load, and cost per transaction at varying volumes, with results summarized in a series of data tables. The study finds that orchestration based sagas provide more predictable failure handling for multi-step retail transactions than pure choreography, that provisioned concurrency meaningfully reduces tail latency for checkout critical functions, and that serverless architectures achieve materially lower cost per transaction than always-on fleets at the load factors typical of retail seasonality, while introducing new considerations around cold start variance, vendor specific quotas, and observability across a larger number of smaller components. The article closes with a discussion of limitations and open research directions, including cross provider portability, cost predictability at extreme scale, and testing strategies for highly distributed function based systems.

Keywords:
serverless computing, Function as a Service, retail transaction processing, event-driven architecture, saga pattern, CQRS, auto scaling, cloud cost optimization, stream processing, distributed systems.

1. INTRODUCTION

Retail organizations operate transaction processing systems that must reconcile two competing pressures. On one side, checkout, payment authorization, inventory reservation, and order confirmation must complete within a few hundred milliseconds to preserve conversion rates, since shoppers abandon carts quickly when a page stalls. On the other side, demand for that same processing capacity swings by an order of magnitude or more between an ordinary Tuesday afternoon and a flash sale, a holiday shopping weekend, or a flash restock of a limited inventory item. Provisioning a fixed fleet of servers for the peak wastes capacity for most of the year, while provisioning for the average risks queueing, timeouts, and lost sales at the peak.

Serverless computing changes the unit of deployment from a long running server process to a short lived function invocation that a cloud provider schedules, scales, and bills per execution. When combined with managed queues, event buses, workflow orchestrators, and serverless data stores, this model allows a retail transaction platform to scale automatically with incoming request volume, pay largely for the compute time actually consumed, and offload much of the operational burden of patching, capacity planning, and fleet management to the cloud provider. These properties make serverless computing an attractive fit for retail transaction workloads, which are characterized by bursty, seasonal, and promotion driven demand.

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

This article organizes the discussion around eight serverless architecture patterns that recur across retail transaction systems: event-driven ingestion, choreography based sagas, orchestration based sagas, fan-out and fan-in parallel checks, CQRS with serverless read replicas, circuit breaker and retry, stream processing for real-time analytics, and API gateway based throttling and shaping of inbound traffic. For each pattern, the article describes the problem it addresses, the component composition typically used to implement it, and the tradeoffs an architect should weigh before adopting it. A reference architecture then combines these patterns into a coherent transaction processing pipeline, and a benchmark study quantifies cold start behavior, throughput scaling, error rates under load, and cost per transaction across a range of simulated retail traffic profiles, including a simulated high volume promotional event.

The remainder of the article is organized as follows. Section 2 provides background on serverless computing and the platforms most commonly used to implement it. Section 3 characterizes the requirements and failure modes specific to retail transaction processing. Section 4 presents the eight architecture patterns in detail. Section 5 proposes a reference architecture. Section 6 describes the benchmark methodology used to evaluate the patterns, and Section 7 presents the resulting performance data. Section 8 analyzes cost, Section 9 addresses security considerations, and Section 10 summarizes three anonymized comparative case studies. Sections 11 through 13 discuss findings, limitations, and future research directions, and Section 14 concludes.

2. Background on Serverless Computing

2.1 Definition and Core Characteristics

Serverless computing refers to a cloud execution model in which the provider dynamically allocates and deallocates compute resources to run application code, and the consumer is billed according to actual resource consumption rather than reserved capacity. The most widely used serverless compute abstraction is Function as a Service (FaaS), in which application logic is packaged as a stateless function that is triggered by an event, such as an HTTP request, a message arriving on a queue, or a change to a database table, and executed within an ephemeral, isolated runtime environment. Baldini et al. describe this model as one in which developers are relieved of the responsibility for provisioning, scaling, and managing servers, at the cost of accepting constraints on execution duration, statelessness, and startup latency.

Serverless architectures generally combine FaaS compute with a set of managed backing services, including object storage, managed message queues, event buses, workflow orchestration services, and serverless databases that themselves scale capacity automatically. Jonas et al. characterize this combination as cloud programming simplified, in the sense that the developer composes managed services and short lived functions rather than operating a persistent server fleet. Roberts and Chapin note that the term serverless is a slight misnomer, since servers still execute the code, but the defining property is that the consumer of the service does not manage those servers directly.

2.2 Evolution from Infrastructure and Platform Services to FaaS

Cloud computing service models are often described using the layered taxonomy formalized by the National Institute of Standards and Technology, which distinguishes Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). FaaS platforms sit above PaaS in the degree of abstraction offered to the developer: rather than deploying an application to a managed runtime that still runs continuously, the developer supplies individual functions that the platform invokes on demand and scales to zero when idle. Hendrickson et al. trace this evolution through early open source FaaS frameworks such as OpenLambda, which sought to formalize the programming model independent of a specific commercial provider. Commercial FaaS offerings became broadly available beginning in the mid-2010s and were followed quickly by managed event routing, workflow orchestration, and serverless database services that completed the surrounding ecosystem needed for production transaction processing.

Villamizar et al. and later comparative studies of monolithic, microservice, and serverless deployment patterns found that decomposing an application into independently deployable and independently scalable units, whether containerized microservices or individual functions, tends to improve resource utilization and deployment agility relative to a monolith, at the cost of increased operational complexity in tracing requests and managing consistency across the resulting distributed system. Retail transaction platforms, which combine many independently scalable concerns such as cart management, pricing, inventory, payment, and fulfillment, are a natural fit for this style of decomposition.

2.3 Serverless Platforms Relevant to Retail Transaction Processing

A retail transaction platform built on serverless principles typically draws on four categories of managed service: function compute, event and message routing, workflow orchestration, and serverless data storage. Table 1

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

summarizes the representative services in each category across the major public cloud providers, as documented in each provider's own developer guides. The comparison focuses on properties most relevant to retail transaction workloads: maximum execution duration, supported concurrency model, and the availability of provisioned or reserved concurrency for reducing cold start impact on latency sensitive paths such as checkout.

Table 1. Representative serverless service categories and platform capabilities relevant to retail transaction processing

Category	Representative Service	Max Execution Duration	Concurrency Scaling Model	Provisioned Concurrency Option
Function compute	AWS Lambda	15 minutes	Automatic, per account concurrency pool	Yes
Function compute	Azure Functions (Consumption plan)	10 minutes (configurable)	Automatic, dynamic worker allocation	Yes (Premium plan)
Function compute	Google Cloud Functions (1st gen)	9 minutes	Automatic, per function instance pool	Limited
Function compute	IBM Cloud Functions (Apache OpenWhisk based)	10 minutes	Automatic, action container pool	No
Event and message routing	Amazon EventBridge / SQS / SNS	Not applicable	Automatic	Not applicable
Event and message routing	Azure Event Grid / Service Bus	Not applicable	Automatic	Not applicable
Workflow orchestration	AWS Step Functions	1 year (Standard workflow)	Automatic	Not applicable
Workflow orchestration	Azure Durable Functions	Unbounded (checkpointed)	Automatic	Not applicable
Serverless data storage	Amazon DynamoDB	Not applicable	Automatic, on-demand capacity mode	Not applicable
Serverless data storage	Amazon Aurora Serverless	Not applicable	Automatic capacity units	Not applicable

Across providers, the FaaS execution model imposes a maximum invocation duration, a stateless execution context that cannot be assumed to persist between invocations, and a cold start penalty when a new execution environment must be initialized to serve a request. McGrath and Brenner measured this cold start penalty directly and found it to vary substantially by language runtime and package size, a finding that later informs the benchmark design in Section 6. Wang et al. further documented that cold start frequency and severity depend on provider specific scheduling and packing behavior that is not fully disclosed, which has direct consequences for architects designing latency sensitive retail checkout paths.

3. Retail Transaction Processing: Requirements and Challenges

3.1 Transaction Volume and Seasonality

Retail transaction volume is not uniform across the year, the week, or the day. Promotional events such as major seasonal sales, flash sales, and limited inventory drops produce short, intense spikes in checkout traffic that can exceed baseline volume by a factor of ten to thirty. Table 2 presents illustrative peak to baseline transaction ratios drawn from published industry retail traffic reporting and used as the load profile basis for the benchmark study in Section 7.

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Table 2. Illustrative peak to baseline transaction volume ratios by retail event type

Retail Event Type	Typical Duration	Baseline Transactions per Second (Index)	Peak Transactions per Second (Index)	Peak to Baseline Ratio
Ordinary weekday	24 hours	1.0	2.5	2.5x
Weekend promotion	48 hours	1.0	6.0	6.0x
Major seasonal sale (single day)	24 hours	1.0	18.0	18.0x
Flash sale (limited window)	1 to 4 hours	1.0	28.0	28.0x
Limited inventory product drop	10 to 30 minutes	1.0	35.0	35.0x
Post outage recovery surge	15 to 60 minutes	1.0	12.0	12.0x

3.2 Consistency, Idempotency, and Exactly Once Effects

A retail checkout transaction typically spans several logically distinct systems of record: a cart or order service, an inventory reservation service, a payment authorization service, a tax and pricing service, and a fulfillment or shipping service. Because FaaS platforms may retry a function invocation after a transient failure, and because message queues used between steps typically provide at least once delivery rather than exactly once delivery, every step in a retail transaction workflow must be designed to be idempotent, meaning that processing the same logical event more than once produces the same result as processing it once. Idempotency is commonly implemented using an idempotency key supplied by the client or generated at ingestion, combined with a deduplication record stored in a low latency key-value store such as a serverless NoSQL table.

Distributed transactions across these services cannot rely on a single relational database transaction, since each service may own its own data store. This motivates the saga pattern, discussed in Section 4, in which a sequence of local transactions is coordinated with compensating actions that undo prior steps if a later step fails, such as releasing an inventory reservation after a payment authorization is declined.

3.3 Latency Requirements by Transaction Stage

Different stages of a retail transaction carry different latency budgets. Interactive stages that block the shopper, such as adding an item to a cart or submitting payment, require sub-second response times to avoid perceived slowness and cart abandonment, while asynchronous stages such as fulfillment routing or analytics event processing can tolerate multi-second or multi-minute completion windows. Table 3 presents representative latency targets by transaction stage used as service level objectives in the reference architecture in Section 5.

Table 3. Representative latency targets by retail transaction stage

Transaction Stage	Interaction Type	Target p50 Latency	Target p99 Latency	Consequence of Breach
Add to cart	Synchronous, interactive	80 ms	300 ms	Perceived lag, minor abandonment risk
Price and promotion calculation	Synchronous, interactive	60 ms	250 ms	Checkout stall
Inventory reservation check	Synchronous, interactive	70 ms	350 ms	Checkout stall, oversell risk if bypassed
Payment authorization	Synchronous, interactive	400 ms	1500 ms	Checkout failure, lost sale
Order confirmation write	Synchronous, interactive	100 ms	400 ms	Duplicate submission risk
Fulfillment routing	Asynchronous	2 s	15 s	Delayed warehouse notification
Order confirmation email or notification	Asynchronous	5 s	60 s	Delayed customer communication

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Analytics and reporting event ingestion	Asynchronous, streaming	10 s	120 s	Delayed dashboard freshness

3.4 Failure Modes in Retail Transaction Systems

Table 4 catalogs the principal failure modes observed in retail transaction processing systems and the architectural pattern from Section 4 most directly addressing each one. This mapping is used to justify the pattern selection in the reference architecture.

Table 4. Common failure modes in retail transaction processing and the primary mitigating pattern

Failure Mode	Typical Cause	Primary Impact	Mitigating Pattern (Section 4)
Duplicate order submission	Client retry, network timeout with server side success	Double charge, oversell	Idempotency key with fan-out/fan-in dedup
Inventory oversell	Race condition between concurrent reservation checks	Fulfillment failure, refund	Fan-out/fan-in with conditional writes
Partial transaction failure	Payment declined after inventory reserved	Held inventory, inconsistent state	Orchestration based saga with compensation
Downstream service overload	Traffic spike exceeding third party payment gateway capacity	Elevated error rate, timeouts	Circuit breaker and retry with backoff
Cold start latency spike	Sudden concurrency increase beyond warm pool	Checkout tail latency breach	Provisioned concurrency, traffic shaping
Message queue backlog	Downstream consumer slower than producer during peak	Delayed order processing	Stream processing with elastic consumers
Cascading retries	Aggressive client or service retry without backoff	Amplified load, secondary outage	Circuit breaker with exponential backoff and jitter
Regional service degradation	Cloud provider zonal or regional incident	Checkout unavailability	Multi-region active-passive failover

4. Serverless Architecture Patterns for Retail Transaction Processing

This section presents eight recurring serverless architecture patterns for retail transaction processing. Each subsection describes the pattern's intent, its typical component composition, and the tradeoffs an architect should weigh. Table 5 provides a consolidated summary before the detailed discussion.

Table 5. Summary of the eight serverless architecture patterns for retail transaction processing

Pattern	Primary Use Case	Key Components	Consistency Model
Event-driven ingestion	Decoupling checkout submission from downstream processing	API gateway, event bus, queue, function	Eventual
Choreography based saga	Multi-step order fulfillment without central coordinator	Event bus, functions subscribing to events	Eventual, per-step compensation
Orchestration based saga	Multi-step order fulfillment with explicit workflow control	Workflow orchestrator, functions as tasks	Eventual, centrally tracked
Fan-out / fan-in	Parallel inventory, pricing, and fraud checks	Event bus or queue, parallel functions, aggregator function	Eventual, joined at aggregation

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

CQRS with serverless read replicas	High volume product catalog and order status reads	Write function, change stream, materialized read store	Eventually consistent reads
Circuit breaker and retry	Protecting checkout from downstream instability	Function middleware, state store for breaker status	Not applicable
Stream processing	Real time analytics and fraud signal aggregation	Streaming service, stream consumer functions	Eventual, windowed
API gateway throttling	Shaping inbound traffic during promotional spikes	API gateway, usage plans, queue based leveling	Not applicable

4.1 Event-Driven Order Ingestion Pattern

In the event-driven ingestion pattern, a checkout submission is accepted at an API gateway, validated by a lightweight function, and published as an event onto a managed event bus or queue rather than being processed synchronously end to end. Downstream functions subscribe to the relevant event types and perform inventory reservation, payment authorization, and order persistence independently. This decoupling allows the ingestion path to remain fast and highly available even when a downstream step, such as payment authorization, experiences elevated latency, because the shopper receives an immediate acknowledgment that the order was accepted for processing rather than waiting on every downstream step.

The tradeoff is that the shopper facing acknowledgment can no longer guarantee that the order will ultimately succeed, since later steps may fail. Retail systems commonly address this by returning an order pending state immediately and pushing a follow up notification, delivered through a web socket, polling endpoint, or push notification, once downstream processing completes or fails.

4.2 Choreography Based Saga Pattern

A saga decomposes a multi-step business transaction, such as place order, reserve inventory, authorize payment, and schedule fulfillment, into a sequence of local transactions, each owned by a different service, with compensating actions defined for each step to undo its effect if a later step fails. In the choreography variant, there is no central coordinator; each function reacts to an event published by the previous step and publishes its own event on completion or failure. Adzic and Chatley describe this style as well suited to serverless deployments because each step can be implemented as an independent function reacting to an event bus topic, with no need to deploy or operate a coordinating service.

Choreography scales well and avoids a single point of coordination, but the overall transaction logic becomes implicit, spread across the event subscriptions of many independently deployed functions, which makes it harder to answer the question of what state a given order is currently in without inspecting logs or building a separate tracking store. As the number of steps in a saga grows, choreography becomes progressively harder to reason about and test end to end.

4.3 Orchestration Based Saga Pattern

The orchestration variant of the saga pattern introduces an explicit workflow definition, executed by a managed workflow orchestration service such as a state machine, that calls each step function in a defined sequence, tracks the current state of the transaction, and invokes compensating functions in reverse order if a step fails. This centralizes the transaction logic in a single, inspectable workflow definition rather than distributing it across independent event subscriptions.

For retail order processing specifically, orchestration based sagas tend to provide more predictable failure handling and simpler operational visibility, since the workflow service typically exposes the current step and history of a given order execution directly, which is valuable both for customer service tooling and for post incident analysis. The tradeoff is a tighter coupling to a specific workflow orchestration service and its execution model, and in some implementations, a per-transition cost that should be included in the cost analysis in Section 8.

Table 6. Comparison of choreography based and orchestration based saga patterns for retail order processing

Dimension	Choreography Based Saga	Orchestration Based Saga
Coordination	Implicit, via event subscriptions	Explicit, via a workflow definition
Operational visibility	Requires custom tracking or log correlation	Native execution history per transaction

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Coupling	Loose between steps	Steps coupled to orchestrator, loose to each other
Scalability of ingestion	High, no coordinator bottleneck	High, workflow service itself scales automatically
Suitability for long sagas (5+ steps)	Harder to reason about as steps grow	Easier to reason about as steps grow
Typical additional cost component	Event bus and queue charges only	Event bus, queue, and workflow state transition charges
Best fit	Small number of steps, high decoupling priority	Multi-step order fulfillment with compensation logic

4.4 Fan-Out / Fan-In Pattern for Parallel Checks

Several checks required before confirming an order, such as inventory availability across warehouses, promotional price validation, and fraud risk scoring, are independent of one another and can be executed in parallel rather than sequentially. The fan-out and fan-in pattern publishes a single incoming event to multiple functions simultaneously, each of which performs one check, and an aggregator function or workflow join step waits for all results before proceeding. This reduces end to end latency relative to running the same checks sequentially, since total latency is bounded by the slowest parallel check rather than the sum of all checks.

A practical concern in the fan-out and fan-in pattern is handling partial completion, such as one check timing out while others succeed. Retail implementations typically bound the aggregation window with a timeout and apply a conservative default, such as treating a fraud check timeout as a manual review trigger rather than an automatic approval, to avoid degrading fraud protection under load.

4.5 CQRS with Serverless Read Replicas

Command Query Responsibility Segregation (CQRS) separates the write path, which validates and persists a change such as a new order, from the read path, which serves queries such as order status lookups or product catalog browsing. In a serverless retail architecture, the write path typically persists to a primary data store optimized for consistency, and a change stream, such as a database change data capture feed, triggers a function that projects the change into one or more read optimized, denormalized stores tuned for the specific query patterns of the storefront, such as full text product search or an order status API.

This pattern is particularly valuable for retail catalog and order status reads, which vastly outnumber writes and have very different access patterns from the write path, allowing the read store to be scaled and indexed independently. The cost is eventual consistency between the write and the projected read store, which is typically acceptable for catalog browsing and order status but requires care for any read that gates a write, such as an inventory check immediately before reservation, which should read from the authoritative store rather than the projection.

4.6 Circuit Breaker and Retry with Exponential Backoff

Retail transaction functions depend on external services, including third party payment gateways, tax calculation services, and shipping rate providers, whose availability and latency the retailer does not control. The circuit breaker pattern tracks the recent error rate of calls to a given downstream dependency and, once that error rate crosses a threshold, short circuits further calls for a cooldown period, returning a fast failure or a fallback response instead of continuing to call a degraded dependency. Combined with retry logic that uses exponential backoff and randomized jitter rather than immediate or fixed interval retries, this pattern prevents a struggling downstream dependency from being further overwhelmed by retry traffic, and prevents that overload from cascading back into the retailer's own function concurrency limits.

In a serverless deployment, circuit breaker state is typically held in a shared, low latency store such as a serverless key-value table, since individual function instances do not share memory and may be recycled between invocations. Table 6 summarizes recommended retry and backoff parameters for the downstream dependencies most common in retail checkout paths.

Table 7. Recommended retry and circuit breaker parameters for common retail checkout dependencies

Dependency	Max Retry Attempts	Base Backoff	Backoff Multiplier	Circuit Breaker Error Threshold	Cooldown Period
Payment gateway authorization	2	200 ms	2.0x	50% over 20 requests	30 s

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Tax calculation service	3	150 ms	2.0x	40% over 20 requests	20 s
Shipping rate provider	3	150 ms	2.0x	40% over 20 requests	20 s
Fraud scoring service	1	100 ms	2.0x	60% over 20 requests	15 s
Inventory service (internal)	2	50 ms	1.5x	50% over 30 requests	10 s
Loyalty and rewards service	2	200 ms	2.0x	60% over 20 requests	30 s

4.7 Stream Processing Pattern for Real-Time Analytics

Beyond completing an individual transaction, retail platforms need near real-time visibility into aggregate metrics such as sales velocity, inventory depletion rate, and fraud signal trends, particularly during promotional events when human operators need to react quickly to emerging issues such as an unexpected sellout or a spike in declined payments. The stream processing pattern publishes each transaction event to a managed streaming service, and a set of stream consumer functions perform windowed aggregation, writing rolled up metrics to a dashboard friendly store.

Because stream consumer functions process records in the order delivered by the streaming service's shards or partitions, throughput scaling requires either increasing the number of shards or ensuring the consumer function itself can keep pace with the assigned shard's throughput, since a single slow consumer can create a backlog for that shard even while other shards are processed promptly. This distinguishes stream processing scaling behavior from the more elastic, per-message scaling of a queue based pattern.

4.8 API Gateway Throttling and Traffic Shaping

During an extreme promotional spike, the limiting factor may not be the retailer's own compute scaling but a downstream dependency's fixed capacity, such as a payment gateway's contracted throughput limit, or the retailer's own desire to protect a shared resource such as a relational inventory database from an unbounded burst. The API gateway throttling pattern applies rate limits and usage plans at the edge, returning an immediate, clearly signaled rejection or a queued response to excess traffic rather than allowing it to propagate into the backend, and pairs naturally with the event-driven ingestion pattern in Section 4.1 to level a traffic burst into a queue that downstream functions drain at a sustainable rate.

5. Reference Architecture

This section combines the patterns from Section 4 into a single reference architecture for a serverless retail transaction processing platform. The architecture is described component by component in Table 7, since the article does not include diagrams; the description below should be read as a left to right data flow, from client request through to fulfillment and analytics.

A shopper initiated checkout request first reaches an API gateway, which applies authentication, request validation, and the throttling pattern from Section 4.8. The gateway invokes an ingestion function that performs minimal synchronous validation, assigns an idempotency key if the client did not supply one, and publishes an order submitted event to an event bus, implementing the event-driven ingestion pattern from Section 4.1. A workflow orchestrator, subscribed to that event, begins an orchestration based saga (Section 4.3) that first invokes a fan-out step (Section 4.4) covering inventory reservation, promotional price validation, and fraud scoring in parallel, then proceeds to payment authorization, guarded by the circuit breaker pattern from Section 4.6, and finally to order persistence and fulfillment routing. Order and inventory state changes are captured by a change stream and projected into read optimized stores using the CQRS pattern from Section 4.5, serving the order status and catalog browsing APIs. In parallel, every transaction event is published to a streaming service for real time analytics, implementing the stream processing pattern from Section 4.7.

Table 8. Reference architecture component inventory and responsibilities

Component	Implements Pattern(s)	Responsibility	Scaling Behavior
API gateway	4.1, 4.8	Authentication, validation, throttling, routing	Automatic, provider managed

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Ingestion function	4.1	Idempotency key assignment, event publication	Automatic, concurrency scaled
Event bus	4.1, 4.2	Durable event routing between services	Automatic, provider managed
Workflow orchestrator	4.3	Saga step sequencing, compensation on failure	Automatic, provider managed
Fan-out check functions	4.4	Inventory, pricing, and fraud checks in parallel	Automatic, concurrency scaled
Payment authorization function	4.3, 4.6	Calls external payment gateway with circuit breaker	Automatic, provisioned concurrency recommended
Order persistence function	4.3	Writes authoritative order record	Automatic, concurrency scaled
Change stream projector	4.5	Projects writes into read optimized stores	Automatic, shard scaled
Read API functions	4.5	Serves order status and catalog queries	Automatic, concurrency scaled
Streaming analytics consumers	4.7	Windowed aggregation for dashboards and fraud trend detection	Manual shard count, elastic consumer scaling
Circuit breaker state store	4.6	Tracks recent error rates per downstream dependency	Automatic, on-demand capacity
Idempotency and deduplication store	4.1, 4.4	Prevents duplicate processing of retried events	Automatic, on-demand capacity

6. Benchmark Methodology

To evaluate the patterns in Section 4 and the reference architecture in Section 5, a benchmark study was conducted using a simulated retail order flow implementing the ingestion, fan-out, orchestration, and circuit breaker patterns described above. Synthetic load was generated at the transaction volume ratios given in Table 2, and results were captured across cold start latency, throughput scaling, error rate under load, and cost per transaction. This section describes the test environment and load generation approach; Section 7 presents the resulting data.

Table 9. Benchmark test environment configuration

Parameter	Configuration
Function runtimes evaluated	Node.js 14, Python 3.9, Java 11, .NET Core 3.1, Go 1.x
Function memory allocation (checkout path)	512 MB to 1024 MB, varied per test
Deployment package size (checkout path)	8 MB to 45 MB, varied by runtime
Workflow orchestration	Standard workflow type, per-transition billing
Primary data store	On-demand capacity key-value store
Load generation tool	Distributed HTTP load generator issuing checkout requests
Load profile	Ramp, sustained peak, and step-spike patterns matched to Table 2 ratios
Region configuration	Single primary region, secondary region for failover tests
Measurement window	10 minute sustained window per test point after ramp-up
Percentile aggregation method	Client observed latency, aggregated per one minute bucket

Two limitations of this methodology are noted for transparency. First, cold start behavior is known to vary over time as cloud providers adjust internal scheduling and packing strategies, a point documented by Wang et al., so

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

absolute cold start figures should be read as illustrative of relative pattern behavior rather than as a permanent benchmark of any specific provider. Second, cost figures in Section 8 are computed from published, publicly available pricing at the time of writing and from the measured invocation counts and durations in this benchmark; they do not include one-time engineering effort or the cost of observability tooling.

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

7. Performance Evaluation

This section presents the data tables resulting from the benchmark methodology in Section 6. Results are organized by cold start behavior, throughput scaling, saga pattern comparison, data store latency, auto scaling responsiveness, and simulated peak event behavior.

7.1 Cold Start Latency by Runtime

Table 10. Measured cold start latency by function runtime, checkout path functions at 512 MB memory allocation

Runtime	Median Cold Start	p90 Cold Start	p99 Cold Start	Warm Invocation p50
Node.js 14	180 ms	340 ms	520 ms	12 ms
Python 3.9	210 ms	390 ms	560 ms	14 ms
.NET Core 3.1	410 ms	780 ms	1150 ms	18 ms
Java 11	890 ms	1600 ms	2400 ms	22 ms
Go 1.x	120 ms	230 ms	360 ms	9 ms

7.2 Throughput Scaling Under Increasing Concurrency

Table 11. Checkout path throughput and latency at increasing concurrent request levels

Concurrent Requests	Achieved Requests per Second	p50 Latency	p99 Latency	Error Rate
50	48	95 ms	310 ms	0.0%
200	192	102 ms	340 ms	0.1%
500	478	118 ms	420 ms	0.3%
1000	941	135 ms	610 ms	0.6%
2500	2310	160 ms	980 ms	1.4%
5000	4480	210 ms	1650 ms	3.2%
10000	8250	290 ms	2900 ms	6.8%

The rising error rate and widening tail latency above roughly 2500 concurrent requests in this test configuration reflect a combination of downstream payment gateway throttling and default account level concurrency limits rather than a fundamental ceiling of the FaaS platform itself, underscoring the importance of the circuit breaker and throttling patterns from Sections 4.6 and 4.8 at high load.

7.3 Choreography Versus Orchestration Saga Latency

Table 12. End to end order completion latency, choreography based saga versus orchestration based saga, five step order flow

Metric	Choreography Based Saga	Orchestration Based Saga
p50 end to end latency	410 ms	455 ms
p99 end to end latency	1350 ms	1180 ms
Compensation trigger latency (failure at step 4)	620 ms	390 ms
Average events published per transaction	9	5
Observed duplicate processing rate (without dedup)	1.8%	0.9%
Time to determine current saga state (operational query)	Requires log correlation, 2 to 5 s	Native, under 200 ms

7.4 Data Store Read and Write Latency

Table 13. Read and write latency comparison, on-demand key-value store versus serverless relational store

Operation	Key-Value Store p50	Key-Value Store p99	Serverless Relational Store p50	Serverless Relational Store p99
Single item read by key	4 ms	12 ms	18 ms	65 ms
Single item write	6 ms	16 ms	22 ms	80 ms

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Conditional write (inventory decrement)	7 ms	19 ms	25 ms	95 ms
Indexed query (order history by customer)	9 ms	28 ms	30 ms	140 ms
Cold capacity scale-up event latency	Not applicable (on-demand)	Not applicable	1.5 s	4.2 s

7.5 Auto Scaling Response Time

Table 14. Time to reach target concurrency after a step increase in request volume

Traffic Step	Baseline to Target Ratio	Time to 50% of Target Concurrency	Time to 95% of Target Concurrency	Requests Affected by Throttling During Ramp
Baseline to weekend promotion	6.0x	8 s	35 s	0.2%
Baseline to major seasonal sale	18.0x	15 s	75 s	1.1%
Baseline to flash sale	28.0x	22 s	110 s	2.6%
Baseline to limited inventory drop	35.0x	26 s	140 s	4.3%

The rising fraction of throttled requests during ramp for the most extreme traffic steps supports provisioning a modest amount of pre-warmed, provisioned concurrency ahead of a known, scheduled event such as a limited inventory drop, since reactive auto scaling alone leaves a measurable window of elevated throttling during the first seconds of an unanticipated or very steep spike.

7.6 Simulated Peak Promotional Event

Table 15. Simulated major seasonal sale event, hourly transaction volume and system behavior over a 24 hour window

Hour of Event	Relative Transaction Volume (Index)	p99 Checkout Latency	Error Rate	Active Function Concurrency (Index)
00:00 (baseline)	1.0	310 ms	0.1%	1.0
06:00 (early ramp)	2.5	340 ms	0.1%	2.6
09:00 (sale opens)	14.0	620 ms	0.8%	13.5
10:00 (peak hour)	18.0	780 ms	1.3%	17.2
12:00 (sustained peak)	15.5	690 ms	1.0%	15.0
18:00 (evening secondary peak)	11.0	560 ms	0.6%	10.8
22:00 (decline)	4.0	380 ms	0.2%	3.9
23:59 (event close)	1.4	320 ms	0.1%	1.5

7.7 Regional Failover Behavior

Table 16. Failover metrics during a simulated primary region degradation

Metric	Observed Value
Time to detect regional degradation (health check based)	18 s
Time to redirect traffic to secondary region (DNS based)	45 s
Checkout requests failed during detection and redirect window	2.9%
Data replication lag at failover, key-value store	1.2 s
Data replication lag at failover, relational store	3.8 s

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Manual intervention required	No, automated runbook triggered failover
Time to full recovery of primary region and traffic rebalancing	22 min

8. Cost Analysis

Serverless cost models bill primarily for actual invocation count and execution duration rather than for reserved capacity, which changes the shape of the cost curve relative to an always-on fleet of virtual machines or containers. This section compares total cost of ownership across three deployment models at the transaction volumes established in Section 3, and breaks down serverless cost by contributing service.

8.1 Total Cost of Ownership Comparison

Table 17. Illustrative monthly total cost of ownership comparison across deployment models at three transaction volume tiers (index cost units)

Transaction Volume Tier	Always-On Virtual Machine Fleet	Container Orchestration Platform	Serverless (FaaS plus Managed Services)
Low (baseline retailer, steady traffic)	100	85	55
Medium (regional retailer with seasonal peaks)	260	210	140
High (national retailer, major promotional peaks)	640	560	410

The relative cost advantage of the serverless model is largest at the low and medium volume tiers, where an always-on fleet must still be sized for occasional peaks that occur a small fraction of the time. At the high volume tier, the advantage narrows because sustained high utilization reduces the idle capacity penalty that always-on models incur, and because per-invocation function overhead and managed service charges become a larger share of total cost. This is consistent with the general cost dynamics described by Adzic and Chatley, who found serverless economics most favorable for workloads with pronounced variability between peak and average load, which characterizes most retail transaction traffic.

8.2 Cost per Transaction at Varying Volume

Table 18. Illustrative serverless cost per transaction (index cost units per 1000 transactions) at varying monthly volume

Monthly Transaction Volume	Function Compute Cost	Managed Queue and Event Bus Cost	Workflow Orchestration Cost	Data Store Cost	Total Cost per 1000 Transactions
1 million	0.42	0.05	0.09	0.18	0.74
10 million	0.39	0.05	0.08	0.15	0.67
100 million	0.35	0.04	0.07	0.12	0.58
1 billion	0.31	0.04	0.06	0.09	0.50

Cost per transaction declines modestly as volume grows, reflecting tiered pricing on several managed services and improved data store cost efficiency at higher utilization, but the decline is far less steep than the corresponding decline in per-unit cost typically seen when amortizing a fixed virtual machine fleet over increasing volume, since serverless pricing is already close to linear in usage.

8.3 Cost Sensitivity to Workflow Orchestration Choice

Table 19. Cost contribution of workflow orchestration under choreography versus orchestration based saga implementation, per 1 million orders

Cost Component	Choreography Based Saga	Orchestration Based Saga
Event bus and queue charges	0.058	0.031
Workflow state transition charges	Not applicable	0.041
Function invocation charges (steps and retries)	0.412	0.365
Total per 1 million orders (index units)	0.470	0.437

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

The orchestration based saga's explicit workflow transition charges are more than offset, in this benchmark, by fewer redundant events and a lower observed retry and duplicate processing rate from Table 8 in Section 7.3, resulting in a slightly lower total cost alongside its operational visibility advantage.

9. Security Considerations

Serverless retail transaction platforms introduce a larger number of independently deployed components than a monolithic or coarsely decomposed system, each of which requires its own access control configuration. This section summarizes the principal threat categories and mitigating controls relevant to a retail transaction platform, and a least privilege access checklist for function level permissions.

Table 20. Principal threat categories and mitigating controls for serverless retail transaction platforms

Threat Category	Example Scenario	Primary Mitigation
Over-privileged function permissions	Order function granted broad data store access beyond its own table	Per-function least privilege roles, one role per function
Injection through event payloads	Malformed event field used unsafely in a downstream query	Schema validation at ingestion, parameterized queries
Payment data exposure	Card data logged or persisted outside a compliant boundary	Tokenization at the edge, no raw card data in function logs
Denial of wallet	Attacker driven traffic spike intended to inflate cloud cost	Rate limiting, budgets and cost alerts, WAF rules at the gateway
Credential and secret sprawl	Database credentials embedded in function environment variables	Centralized secrets manager with per-function scoped access
Insecure inter-service event trust	A function accepting event bus messages without verifying source	Event source validation, signed or authenticated event envelopes
Dependency supply chain risk	Vulnerable third party package bundled into a function deployment package	Automated dependency scanning in the build pipeline
Insufficient audit trail across many small functions	Difficulty reconstructing an incident across dozens of functions	Centralized structured logging with correlation identifiers

Table 21. Function level least privilege access control checklist

Checklist Item	Recommended Practice
Role granularity	One execution role per function, not shared across functions
Data store scope	Grant access only to the specific tables or partitions the function reads or writes
Network scope	Restrict outbound calls to the specific dependencies the function requires
Secret scope	Grant access only to the specific secret entries the function requires, not the entire secrets store
Write versus read separation	Read path functions should not hold write permissions to the authoritative order store
Time bound elevated access	Any temporary elevated access for operational tasks should expire automatically
Logging of denied access attempts	Access control denials should be logged and alertable, not silently dropped

10. Comparative Case Studies

This section summarizes three anonymized, generalized case studies drawn from publicly discussed retail serverless adoption patterns, presented without identifying any specific organization, to illustrate how the patterns in Section 4 apply across different retail contexts. Each case study is described in terms of the transaction volume profile it addresses and the patterns it emphasizes.

Table 22. Comparative summary of three generalized retail serverless adoption profiles

Dimension	Profile A: Flash Sale Specialist	Profile B: Omnichannel Grocery	Profile C: Marketplace Aggregator
Dominant traffic pattern	Extreme, short duration spikes	Steady with moderate weekend peaks	Continuous, multi-seller variability

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

Peak to baseline ratio	30x or higher	3x to 5x	Variable, 5x to 15x depending on seller mix
Primary saga style	Orchestration based	Orchestration based	Choreography based (per-seller isolation)
Emphasis on provisioned concurrency	High, for known sale start times	Low, gradual ramps are predictable	Medium, varies by seller campaign timing
Primary read scaling concern	Catalog reads during pre-sale browsing	Store and delivery slot availability reads	Cross-seller catalog aggregation reads
Most cited operational challenge	Cold start variance at exact sale start second	Multi-region inventory consistency across stores	Per-seller circuit breaker isolation

Across all three profiles, the fan-out and fan-in pattern for parallel inventory, pricing, and fraud checks (Section 4.4), and the circuit breaker pattern for downstream dependency protection (Section 4.6), were consistently applied regardless of traffic shape, while the choice between choreography and orchestration for the core saga, and the degree of investment in provisioned concurrency, varied according to how predictable and how extreme the traffic pattern was for that specific retail context.

11. Discussion

The benchmark results in Section 7 and the cost analysis in Section 8 support three general conclusions for architects evaluating serverless computing for retail transaction processing. First, the choice between choreography and orchestration for the core order saga should be driven primarily by the number of steps and the operational visibility required, rather than by cost alone, since the cost difference between the two styles was modest in this study while the operational visibility difference was substantial. Second, provisioned concurrency for checkout critical functions is not optional at the traffic ratios typical of major promotional events; the throttling observed during ramp in Table 11 (Section 7.5) indicates that reactive automatic scaling alone leaves a measurable window of degraded service at the most extreme traffic steps, and this window grows with the steepness of the step. Third, the cost advantage of serverless architectures over always-on fleets, summarized in Table 12 (Section 8.1), is largest precisely for the retailers with the most pronounced seasonality, which is also the population for whom checkout resilience during a peak matters most, making the combination of cost and resilience benefits mutually reinforcing rather than a straightforward tradeoff.

The results also highlight a consideration less discussed in general serverless literature but particularly relevant to retail: the interaction between the retailer's own auto scaling and a third party payment gateway's fixed capacity, observed in the throughput scaling data in Table 8 (Section 7.2). A retail platform can scale its own compute layer far faster than many payment processors can absorb additional authorization volume, which means the circuit breaker and throttling patterns in Sections 4.6 and 4.8 function not only as resilience mechanisms for the retailer's own system but as a form of considerate load shaping toward shared external dependencies.

12. Limitations

- Benchmark figures in Section 7 were captured on a single provider's platform configuration and a single test period; cold start behavior in particular is known to shift over time as providers adjust internal scheduling, so absolute figures should be treated as illustrative of relative pattern behavior rather than as durable benchmarks.
- Cost figures in Section 8 reflect published pricing at the time of writing and the specific invocation and duration profile measured in this study; actual cost for a given retailer will depend on its specific traffic shape, payload sizes, and negotiated commitments with its cloud provider.
- The case studies in Section 10 are generalized composites drawn from publicly discussed adoption patterns rather than a controlled study of specific named organizations, and should be read as illustrative rather than as a systematic industry survey.
- The study focuses on a single cloud provider's ecosystem for the reference architecture and benchmark; cross provider portability, and the operational cost of building a multi-provider serverless retail platform, are not evaluated here.
- Testing methodology and tooling for highly distributed, event-driven serverless systems remains less mature than for monolithic or traditional microservice systems, and the benchmark in this study relied on external, black box load testing rather than fine grained internal tracing of every function boundary.

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

13. Future Research Directions

- Cross provider portability of the saga, fan-out and fan-in, and CQRS patterns, including the practicality of abstraction layers that reduce lock-in without sacrificing the provider specific performance optimizations documented in this study.
- Cost predictability tooling that forecasts serverless spend under a projected promotional traffic profile with the same confidence retailers currently place in capacity planning for fixed fleets.
- Automated, workload aware selection of provisioned concurrency levels ahead of scheduled promotional events, informed by historical traffic shape rather than static, manually configured warm pools.
- Improved integration testing strategies for orchestration based sagas spanning many independently deployed functions and third party dependencies, including consistent approaches to simulating downstream payment gateway degradation during pre-event load testing.
- Longitudinal study of cold start behavior across providers over multi-year periods, to establish whether the improvements observed by researchers such as Wang et al. and others continue at a pace that reduces the need for provisioned concurrency over time.

14. Conclusion

Serverless computing, applied through the combination of event-driven ingestion, saga based order coordination, parallel fan-out checks, CQRS read scaling, circuit breaker resilience, stream processing analytics, and API gateway traffic shaping, provides a coherent and cost effective foundation for retail transaction processing systems that must absorb large, seasonal swings in demand. The benchmark data presented in this article indicate that these patterns, properly combined in a reference architecture and paired with deliberate use of provisioned concurrency for checkout critical paths, can sustain acceptable latency and error rates through promotional traffic spikes exceeding thirty times baseline volume, while achieving lower total cost of ownership than always-on alternatives, particularly for retailers with pronounced seasonality. The remaining open questions, concentrated around cross provider portability, cost forecasting, and testing methodology for highly distributed function based systems, represent practical next steps for both practitioners adopting these patterns and researchers studying the broader serverless computing paradigm.

REFERENCES

1) Adzic, G., and Chatley, R. (2017). Serverless computing: economic and architectural impact. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 884 to 889. Association for Computing Machinery.
2) Amazon Web Services. (2021). AWS Lambda Developer Guide. Amazon Web Services technical documentation.
3) Amazon Web Services. (2021). AWS Step Functions Developer Guide. Amazon Web Services technical documentation.
4) Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., and Suter, P. (2017). Serverless computing: current trends and open problems. In Research Advances in Cloud Computing, pp. 1 to 20. Springer.
5) Baldini, I., Cheng, P., Fink, S. J., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P., and Tardieu, O. (2017). The serverless trilemma: function composition for serverless computing. In Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pp. 89 to 103. Association for Computing Machinery.
6) Baresi, L., and Filgueira Mendonca, D. (2019). Towards a serverless platform for edge computing. In 2019 IEEE International Conference on Fog Computing (ICFC), pp. 1 to 10. IEEE.
7) Bermbach, D., Karakaya, A. S., and Buchholz, S. (2020). Using application knowledge to reduce cold starts in FaaS services. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, pp. 134 to 143. Association for Computing Machinery.
8) Elgamal, T. (2018). Costless: optimizing cost of serverless computing through function fusion and placement. In 2018 IEEE/ACM Symposium on Edge Computing (SEC), pp. 300 to 312. IEEE.
9) Eyk, E. van, Iosup, A., Seif, S., and Thommes, M. (2017). The SPEC cloud group's research vision on FaaS and serverless architectures. In Proceedings of the 2nd International Workshop on Serverless Computing, pp. 1 to 4. Association for Computing Machinery.
10) Fan, Y., Lu, J., and Zhang, H. (2020). Improving serverless application performance through predictive function placement. IEEE Cloud Computing.
11) Fowler, M. (2015). Microservices: a definition of this new architectural term. martinfowler.com.

 © Author(s) | CC BY 4.0 International License.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
https://ijetrm.com/

12) Fox, G. C., Ishakian, V., Muthusamy, V., and Slominski, A. (2017). Status of serverless computing and function as a service (FaaS) in industry and research. arXiv preprint arXiv:1708.08028.
13) Google Cloud. (2021). Cloud Functions documentation. Google Cloud technical documentation.
14) Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2016). Serverless computation with OpenLambda. In 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16). USENIX Association.
15) Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., and Patterson, D. A. (2019). Cloud programming simplified: a Berkeley view on serverless computing. arXiv preprint arXiv:1902.03383.
16) Kaviani, N., Kalinin, D., and Maximilien, M. (2019). Towards serverless as commodity: a case of Knative. In Proceedings of the 5th International Workshop on Serverless Computing, pp. 13 to 18. Association for Computing Machinery.
17) Kritikos, K., and Skrzypek, P. (2018). A review of serverless frameworks. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp. 161 to 168. IEEE.

 © Author(s) | CC BY 4.0 International License.