International Journal of Advanced Engineering Science and Information Technology (IJAESIT)

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

# Serverless Cloud Computing for Large-Scale E-Commerce Transaction Processing

**Anumandla Mukesh**

Independent Researcher, India

mukeshanumand@gmail.com

**ABSTRACT:** E-commerce architectures begin to leverage serverless cloud computing. An objective, evidence-based analysis names key workload classes, establishes suitability for Function-as-a-Service patterns, and articulates a checklist for design decisions. Latency-sensitive, durable, orchestrated transaction processing is feasible; cost-sensitivity, scale variability, and requirement isolation favour serverless paradigms. Supporting foundations are in place, yet service availability, compliance factors, and nonnative data-consistency patterns warrant caution.

Serverless computing offers a compelling option for a variety of applications. Yet when applied to cloud-native transaction processing—especially in critical, large-scale e-commerce settings—the modality faces specific challenges of non-determinism, statefulness, and acute latency sensitivity. Naïvely evaluated against four sources of cost and reliability concern, serverless approaches are nonetheless indicated for three primary classes of orders: low-cost commodity fulfilment and fraud detection, where lower reliability appears acceptable; durable, low-frequency orchestrated workflows; and typical sessions, augmented with caching and sharding for scaling.

**KEYWORDS:** Serverless Cloud Computing, Function-as-a-Service (FaaS), Cloud-Native E-Commerce Architectures, Transaction Processing Systems, Latency-Sensitive Workloads, Durable Orchestrated Workflows, Cost-Efficient Cloud Scaling, Elastic Scalability Patterns, Stateful versus Stateless Design, Non-Deterministic Execution Challenges, Data Consistency in Serverless Systems, Caching and Sharding Strategies, Compliance in Cloud Transaction Systems, Reliability Engineering in Distributed Systems, Fraud Detection Workloads.

## I. INTRODUCTION

Serverless Cloud Computing for Large-Scale E-Commerce Transaction Processing defines research questions, objectives, scope, and significance; articulates criteria for evaluating serverless suitability in large-scale transactions. Large enterprise-class e-commerce platforms typically generate millions of transactions every day. These transactions must be processed reliably within well-defined time periods to ensure good customer experiences. Customers expect servers to be faster, cheaper, and more reliable than before. Achieving this is very challenging, especially for transaction-management subsystems. Asynchronous invocation provides an opportunity to process transactional workloads using serverless cloud computing. Serverless cloud-computing architecture provides multiple advantages versus traditional counterparts; at the same time, it poses unique challenges. Cost, latency, scalability, and reliability are the primary considerations for large-scale systems hosted in a public cloud environment. It is important to examine how suitable the serverless paradigm is for processing mission-critical e-commerce transactions.

The suitability of serverless cloud architecture for processing large-scale e-commerce transactions is examined in this section. Analyses of the patterns, workloads, and architectures of typical e-commerce transaction subsystems provide the basis for this examination. The focus is specifically on transactional workload subsystems such as payment processing, fraud detection, guarantee management, and so on and not on the entire e-commerce cloud-hosted application. In addition to suitability, the considerations identified in the preceding two paragraphs are also useful for understanding other aspects—the strengths, weaknesses, risks, and pitfalls—of serverless cloud architecture applied to large-scale transaction workloads.

### 1.1. Purpose and Scope of the Study

The purpose of this study is to evaluate the suitability of serverless cloud computing for large-scale e-commerce transaction processing. The definition of large-scale emphasizes the interplay among transaction wrap-up latency, transaction volume, and supporting capacity of two external units: transaction-service databases and payment-processing services, whether in-house or third-party. The suitability criteria use operational characteristics identified in a review of state-of-the-art SC²T architectures for service-based processing—such as event-driven processing by function-as-a-services (FaaS); the stateless, idempotent, scale-out properties of functions; and cloud providers' reliance

International Journal of Advanced Engineering Science and Information Technology (IJAESIT)
|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|
DOI: 10.15662/IJAESIT.2021.0406002

on elasticity defined by demand—against requirements of large-scale transaction processing. These considerations encompass trade-offs typically associated with evolving capital expenditures into operational expenditures, specifically relative to latency, cost, scalability, and reliability.

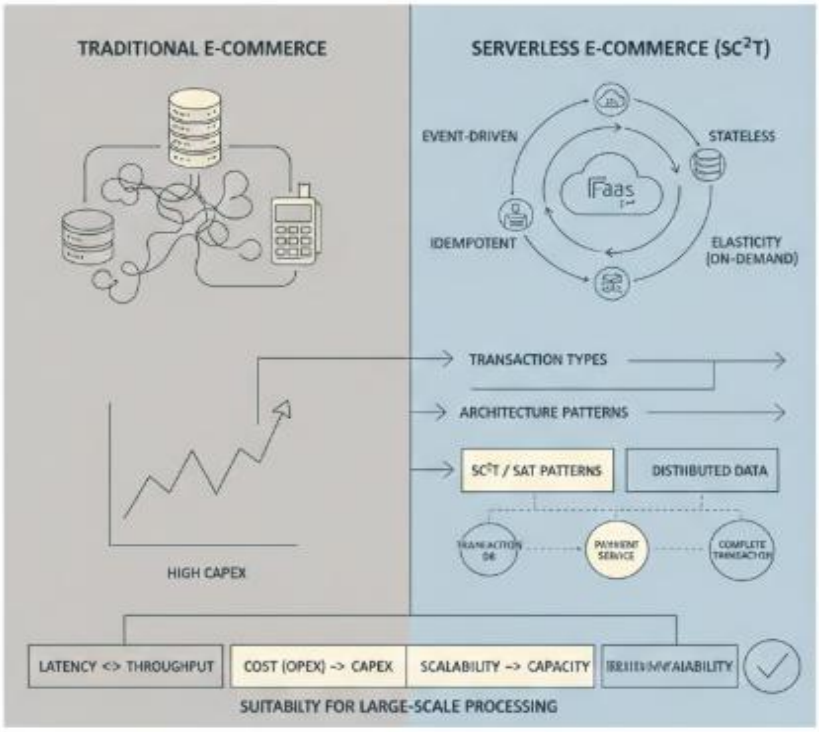
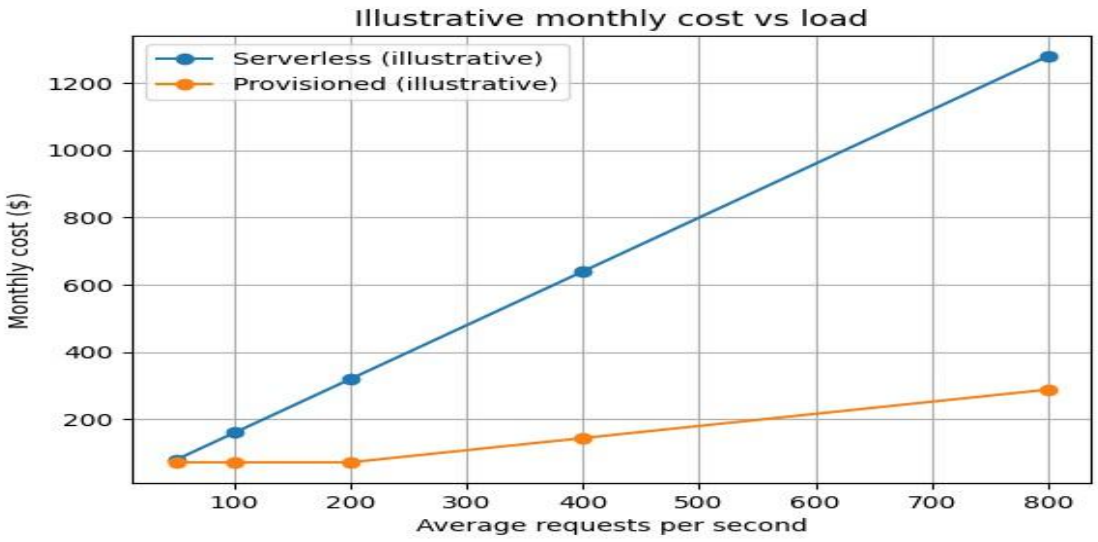



**Fig 1: Evaluating Serverless Suitability for High-Volume E-Commerce: Architectural Patterns, Elasticity**

**Trade-offs, and Distributed Data Management in SC²T Frameworks**

Support for serverless SC²T workloads is examined on three fronts. Transaction types and event correlations are described—the most salient factors in supporting concentrated volumes, minimizing latency, and achieving sufficient throughput across the entire system. Architecture patterns appropriate for transaction-service workloads are discussed next; SC²T and SAT patterns are explicitly evaluated against serverless alignment. Finally, distributed data-management tactics are surveyed with an emphasis on those supporting enterprise-level serverless elasticity within a cloud region.



|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

**Equation 1) Cost model (serverless "pay only for execution duration / TET")**
**Step-by-step derivation**
**Step 1 — Define per-request billable components**
Let each transaction request trigger **k functions** (or k billable serverless steps).
For step *i*:

- execution time: $t_i$ (seconds)
- memory allocated: $m_i$ (GB)
- request count: 1 per invocation

Typical serverless billing is well captured by:

$$C_i = C_{\text{req}} + C_{\text{compute}}(m_i, t_i)$$

A common compute form is GB-seconds pricing:

$$C_{\text{compute}}(m_i, t_i) = p_{\text{GBs}}\,(m_i\, t_i)$$

So for one end-to-end transaction:

$$C_{\text{txn}} = \sum_{i=1}^{k} \left(p_{\text{req}} + p_{\text{GBs}}(m_i t_i)\right)$$

**Step 2 — Convert to per-unit-time cost**
Let arrival rate be $\lambda(t)$ transactions/second (the paper emphasizes temporally non-uniform workloads / elasticity).
Then total cost over time horizon $T$ is:

$$C(T) = \int_0^T \lambda\,(t)\, C_{\text{txn}}\, dt$$

If load is approximately constant $\lambda$:

$$C(T) = \lambda\, T\, C_{\text{txn}}$$

**Step 3 — Compare to provisioned servers (baseline)**

If provisioned capacity requires $N$ instances, each at hourly cost $p_{\text{hr}}$, then over $T$ hours:

$$C_{\text{prov}}(T) = N\, p_{\text{hr}}\, T$$

## II. BACKGROUND AND MOTIVATION

Consequently, use cases requiring elasticity, i.e. temporally non-uniform workloads with variable scaling ratios, are critical for any resource-purchase decision. Application architectures with serverless or Function-as-a-Service deployments have become one of the most popular approaches for concurrent execution of a large number of short-lived tasks. With unmatched scaling characteristics, these paradigms can run arbitrarily high transaction loads in theory. However, transaction-processing systems deployed on such architectures, or portions thereof, have been shown to incur long latency. Latency, along with cost, is arguably the most important metric considered when evaluating serverless resource usage for transaction processing. The second important set of metrics are the long-term Elasticity and Reliability trade-offs. Testing the suitability of serverless Function-as-a-Service architectures for large-scale transaction-processing systems is thus timely as enterprise workloads evolve.

Cloud service providers offer serverless Function-as-a-Service computing platforms with stateful capabilities. These offerings simplify the development of serverless applications without requiring users to think about Distributed Systems design. While platforms are marketed as support for event-driven application development, high level workflows are still considered suitable for serverless-based cloud-level transaction processing. Supporting analysis assumes that models present sufficient guarantees for Stateful Functions (SaaS) invoked through orchestration but without assessing or mitigating additional overhead imposed for such constructions, including correctness under serverless and IaC constraints.

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

### 2.1. Key Considerations for Serverless Architectures in E-Commerce

Serverless concepts, Cloud Services, and Trade-Offs

The notion of serverless cloud computing is relatively new, and some of its concepts are still being refined. Function-as-a-Service (FaaS) models distinctively underpin serverless computing. Cloud service providers execute application code hosted in the cloud on behalf of customers. The customer simply specifies the code, its cloud storage location, input parameters, and, on execution, pays only for the computing duration. Providers usually start anew for each code invocation, but may optimize performance by reusing warmed-up run-time environments. In addition to FaaS, cloud providers offer non-FaaS services that can be instantaneously invoked, billed on a per-request basis, and automatically rescaled. Such services, too, are serverless in the colloquial sense. Trade-offs associated with any paradigm apply here as well, hence suitability for a large-scale transaction-processing workload requires further analysis.

Hypothetical costs for an e-commerce transaction processing system hosted on a major cloud provider with vendor-verifiable load characteristics help assess the suitability of a serverless architecture. Costs reflect the underlying resource usage, but together with modelling considerations provide insensitive order-of-magnitude cost approximations. Variations in modelled cost effects are more impactful than accuracies in underlying parameters. Latency, rate limitation, scalability, and reliability are other relevant factors. Vendor-latency guarantees are inadequate for time-sensitive transactions, however, since average, rather than worst-case, latencies are exposed, and long-distance invocations incur extra delays.

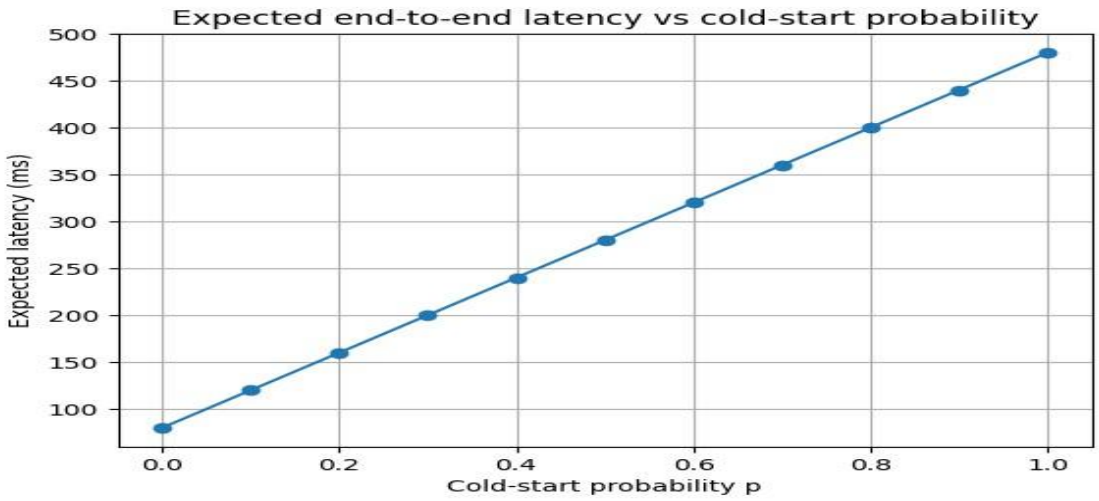
| Avg RPS | Requests/month (M) | Serverless cost ($) [illustrative] | Provisioned cost ($) [illustrative] |
|---|---|---|---|
| 50 | 129.6 | 79.92 | 72.0 |
| 100 | 259.2 | 159.84 | 72.0 |
| 200 | 518.4 | 319.68 | 72.0 |
| 400 | 1036.8 | 639.36 | 144.0 |
| 800 | 2073.6 | 1278.72 | 288.0 |

## 3. Serverless Architectures for E-Commerce

Function-as-a-Service (FaaS) is the function-level serverless variant of the Function-as-a-Service (FaaS) paradigm. Running codes in interest-based small, limited stateful micro-services by cloud vendors work as a script under policy defined parameters that are triggered by an event. These functions can be integrated or connected with their own API or 3rd-party API (services like payment, email, SMS etc) or services under the offered execution platform for defined use cases. Stream-based, publish-subscribe (event-driven) processing models are popular for this Interest Computing paradigm. Events, triggers or hops are used to integrate various sources or mechanisms behind a solution like source of information, source of action or decision supporting mechanism in a single logical unit where functions running in parallel are fed with events or data whenever chance arises. Cloud services offer running code without managing servers using the concept of a basic unit (used – or needed in execution) which is able to respond to HTTP requests (API). The service is mainly pushed as a low-cost solution where the vendor charges Task Execution Time (TET), which is very low against the cost of provisioning and managing a server. Key factors play a vital role in minimizing the cost and performance more focus on critical, less-scalable service for business and industries where cost of downtime is very high.

Implementation and correctness validation of computing above Interest Computing not on FaaS in general but on a leading Cloud Vendor's FaaS—dream compute are evaluated based on SLA. State management, session handling, retries and correctness validation behaviour of Interest Computing coded and implemented on Interest Computing Architecture (ICA). FaaS introduces Event-Driven Computing (EDC) for application load where EDC does not maintain any internal system state; on event (data) arrival, function is loaded (WARM Start) and procedure executed. Application can request 3rd-party APIs and function is kept idle at cloud vendor during past or free time. Cost and timing metrics of Digital-Injury-Examined (DIE) project considering (base level) injury-examined-time are out for now and currents product is live.

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|
DOI: 10.15662/IJAESIT.2021.0406002




**Equation 2) Latency model (warm vs cold start + I/O + queueing)**
**Step-by-step derivation**
**Step 1 — Decompose end-to-end latency**
For one transaction:

- trigger/ingress overhead: $L_{\text{ingress}}$
- function compute + internal logic: $\sum_i L_{\text{fn},i}$
- external I/O (DB, payment provider, queues): $\sum_j L_{\text{io},j}$
- queue wait due to throttling/quota: $W_q$
- cold-start penalty (only if cold): $L_{\text{cold}}$

So:

$$L \;=\; L_{\text{ingress}} + \sum_i L_{\text{fn},i} + \sum_j L_{\text{io},j} + W_q \;+\; I_{\text{cold}}\, L_{\text{cold}}$$

where $I_{\text{cold}} \in \{0,1\}$.

**Step 2 — Expected latency using cold-start probability**

Let $p = P(I_{\text{cold}} = 1)$. Then:

$$\mathbb{E}[L] \;=\; L_{\text{warm}} \;+\; p\, L_{\text{cold}}$$

where $L_{\text{warm}} = L_{\text{ingress}} + \sum_i L_{\text{fn},i} + \sum_j L_{\text{io},j} + W_q$.

### 3.1. Function-as-a-Service and Event-Driven Processing

Function-as-a-Service supports fine-grained event-driven processing via invocations in the microsecond range, driven by application-managed schedulers. A key serverless concept is that resources are made available only for code execution and remain idle at all other times. Therefore, runtime hosting costs are incurred for every transient function execution. Clouds maintain pools of resource quotas and automatically scale usage; when quota capacity is exceeded, an external queue absorbs events. Classical serverless computing, built with specific conditions and system designs in mind, assumes stateless execution and idempotent transformations.

Function execution is stateless and without side effects. External resources can be accessed (read and written) but must not be modified in between two function invocations. These resources could experience considerable resource contention—hence the need for distribution and replication. It is not mandatory for transactional workloads to be idempotent at the function level (although this is a necessary condition for guaranteeing safety when using an external ondemand service); however, each invocation must be each treated as independent of the others. Aspects such as fault tolerance, reliability, persistence, consistency, availability, and protection against duplication must be guaranteed by actions that are executed as a whole on other resources (e.g. database, message queue).

International Journal of Advanced Engineering Science and Information Technology (IJAESIT)
|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|
DOI: 10.15662/IJAESIT.2021.0406002

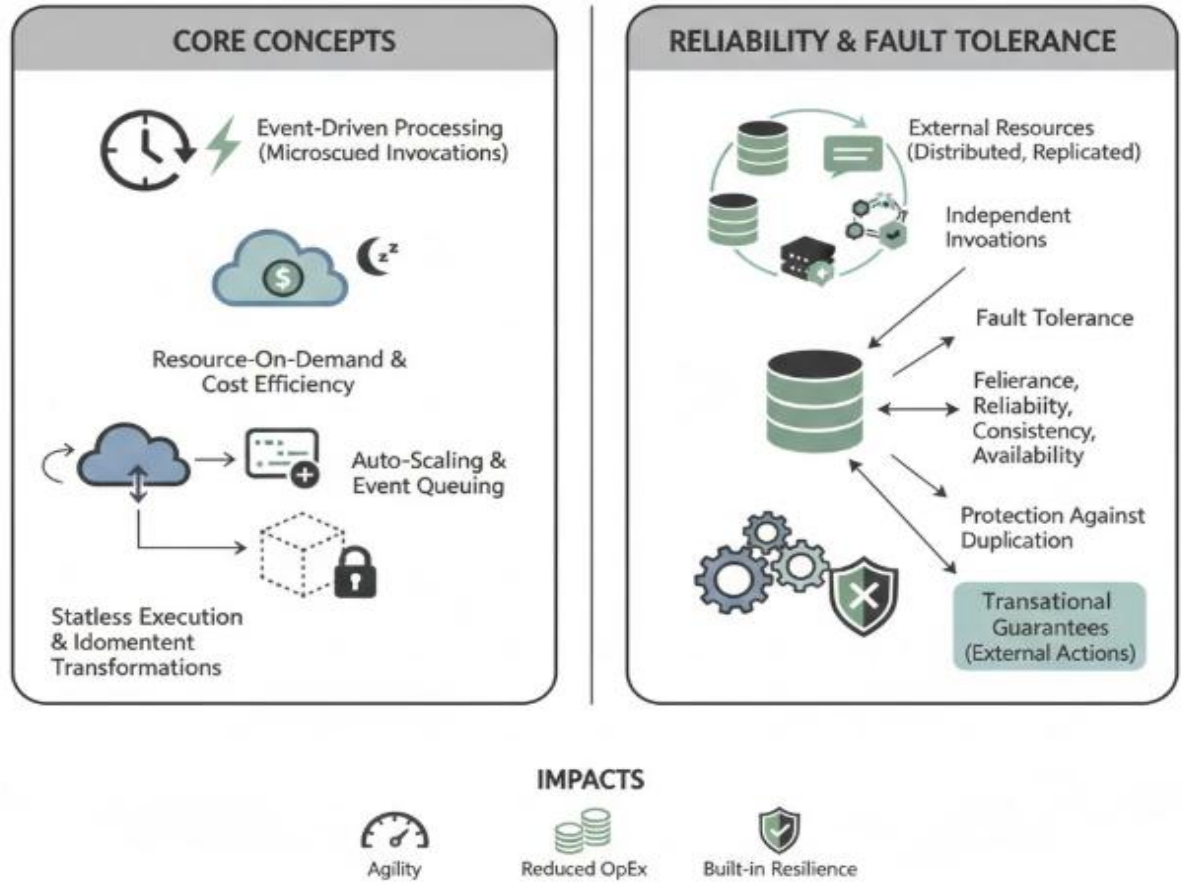



**Fig 2: Transient Logic, Persistent Guarantees: Statelessness and Idempotency in Function-as-a-Service (FaaS)**

### 3.2. Statelessness, Idempotence, and Scaling Characteristics

Serverless execution is essentially stateless and scales indefinitely. These characteristics ease provisioning, shrinking, and scaling up of state-centric services such as databases and caches. Serverless functions do not maintain their own session or transaction scope. Any data structure must therefore be stored externally, typically in a long-lived, low-latency, distributed key/value datastore for fast access. The integrated storage solutions of the cloud providers (DynamoDB, BigTable, CosmosDB, etc.), designed for horizontal partitioning, replication, and failure recovery behind a low-latency API, are particularly suitable because they can be accessed without additional use-discovery logic. The distribution of such storage is largely transparent to the users; they just specify a key.

Because databases are stateless for each request, they are usually COPY of the writing scale, and for function-as-a-service deployments may have a large number of cold starts (when a new instance is created after being idle for a while) when specific instances are needed. Functions communicating with a low-latency stateful source (for example, a key-structure brick or a session store) would have a CTRL more significant than that of functions that query one copy of the state and thus change fast but with substantial CTRL. In addition, any such session stop must maintain a long foundation's delay when running MANAGED, as maintaining the session stops at each request's start can also be time-consuming.

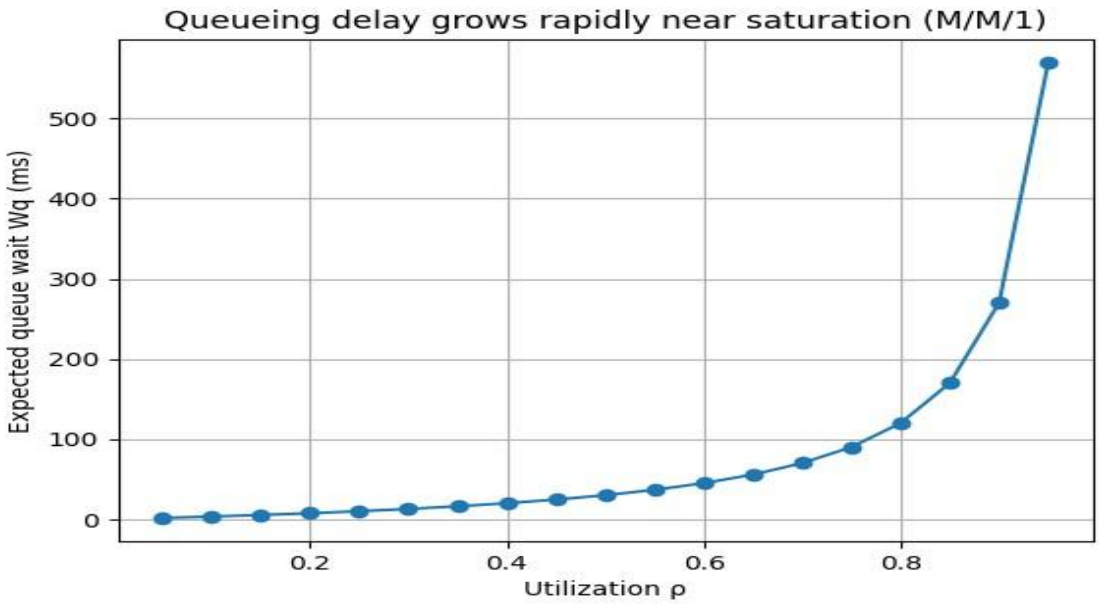
## IV. TRANSACTION PROCESSING WORKLOADS IN LARGE-SCALE E-COMMERCE

E-commerce transactions typically rely on multiple, sometimes complex workflows to process payments, update inventory, manage guarantees, detect fraud, enforce compliance, and so on. A taxonomy of processing characteristics is useful for understanding compatibility with serverless patterns: controls that must be executed in a strictly ordered sequence with low latency and high throughput (e.g., payment processing) are the most demanding; those that can remain eventual in nature (e.g., fraud detection) place much less stress on the system; and others (e.g., guarantee management) fall somewhere in between. Each category can thus be examined against a common set of criteria, which address reliability, data consistency, latency and throughput requirements, and extra-functional properties (e.g., adherence to security or legal controls).

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

Order placement and inventory management are among the most critical workflows; they must be performed together to preserve an ACID-like guarantee that no new order is accepted unless the associated stock is guaranteed. The second pattern is essentially the BASE alternative of the first, since the probability of making a cross-transaction conflict must remain sufficiently low until a new inventory forecast is provided. Payment processing must enforce business controls defined by regulation and law, and thus require an additional set of audit trials together with rigidity of execution; fraud detection must comply with these controls, but is typically executed asynchronously, and thus subject to a BASE-like guarantee.




**Equation 3) Queueing / throttling model (why latency "blows up" near saturation)**
**Step-by-step derivation (simple M/M/1 approximation)**
**Step 1 — Model**

- arrival rate $\lambda$ (req/s)
- service rate $\mu$ (req/s) (how fast the system completes requests)
- utilization $\rho = \lambda/\mu$

**Step 2 — Expected waiting time in queue**
For M/M/1:

$$W_q = \frac{\rho}{\mu(1-\rho)}$$

**4.1. Order Placement and Inventory Management**
An e-commerce transaction is initiated by a customer placing an order. Order placement typically requires the submission of an order model that includes at least a reference to the customer, a customer shipping address, and a list of line items. Each line item contains a reference to a product, a requested quantity, and optionally, a description of the item in the order. The order placement workflow first verifies that the items in the cart have not already been taken by other customers after the customer started checking out the cart. After this check passes, the inventory of the ordered item(s) is locked. Now that it is not possible for other customers to purchase the same products starting from the time the request arrives, inventory can be quickly checked without a lengthy database or remote service call. The payment processing component is then invoked.

The transaction management system needs to be aware of business constraints such as the fact that an order cannot be placed unless there are enough products in stock, and therefore the inventory needs to be managed accordingly.The inventory component fulfills the item-locking function. When inventory is locked by an order, it remains in lock state for a specified duration. During this time window, orders can still be placed, but only if the quantity of the product requested in the order is less than or equal to the quantity of the product that has been locked by other orders.

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

| Cold-start probability p | Expected latency E[L] (ms) |
|---|---|
| 0.0 | 80.0 |
| 0.1 | 120.0 |
| 0.2 | 160.0 |
| 0.3 | 200.0 |
| 0.4 | 240.0 |
| 0.5 | 280.0 |
| 0.6 | 320.0 |

### 4.2. Payment Processing and Fraud Detection

Payment processing is integral to e-commerce transaction systems and it necessitates strict compliance. Card payments are subject to the Payment Card Industry Data Security Standard (PCI DSS), which mandates stringent security controls over cardholder data handling, storage, and transmission, along with regular auditing. Card payments also integrate anti-fraud checks, where card-not-present fraud detection is particularly difficult due to the absence of physical verification (e.g., signature), which makes ineffective measures like CAPTCHA and 3-D Secure update very hard for legitimate customers. Executive fraud detection extends beyond monitoring payments to undermining an entire e-commerce business. Regulation requires a refund operation similar to real-time payments.

These various requirements and use cases dictate how payments integrate, manage, share data, and communicate with external parties. Compliance regulations create implications for every payment operation, from data collection and usage through to worker behaviour, while the multiple external integrations introduce additional requirements such as monitoring, centralization, and shared experience.

## V. ARCHITECTURAL PATTERNS FOR SERVERLESS TRANSACTION SYSTEMS

E-commerce transaction processing systems have well-established implementations and behavior patterns. Some workflows within transaction-processing systems are orchestrated, sensibly implemented with traditional process-management tools. Other transaction-processing tasks are more straightforward: the processing takes place in response to events or state alterations, and control only needs to flow through a well-defined sequence of steps. Such patterns—including payment processing, many fraud-detection mechanisms, and external-integrations management—can be implemented with serverless technologies to avoid the operational burden of keeping a continuously running server.

The durability of the process state is relevant. Some serverless orchestrations and durable function implementations maintain execution state in serverless forms; other implementations provide external state management. Event-driven architectures can be advised over orchestration whenever possible; when they're not, orchestration is usually preferable to physical-process management. Event sourcing, combined with Command Query Responsibility Segregation (CQRS) principles, can be particularly useful. Serverless technologies for event stores exist, though synchronization and updating can be challengingly costly, given the distributed-data storage and caching recommendations.

**Equation 4) Inventory locking & ACID vs BASE: conflict probability vs lock duration**

Step-by-step derivation (Poisson overlap approximation)

**Step 1 — Define "contention" rate**

For a hot SKU, let competing checkouts arrive as a Poisson process with rate $\lambda_c$ per second.

**Step 2 — Define lock window**

Lock duration is $\tau$ seconds (the paper: inventory "remains in lock state for a specified duration").

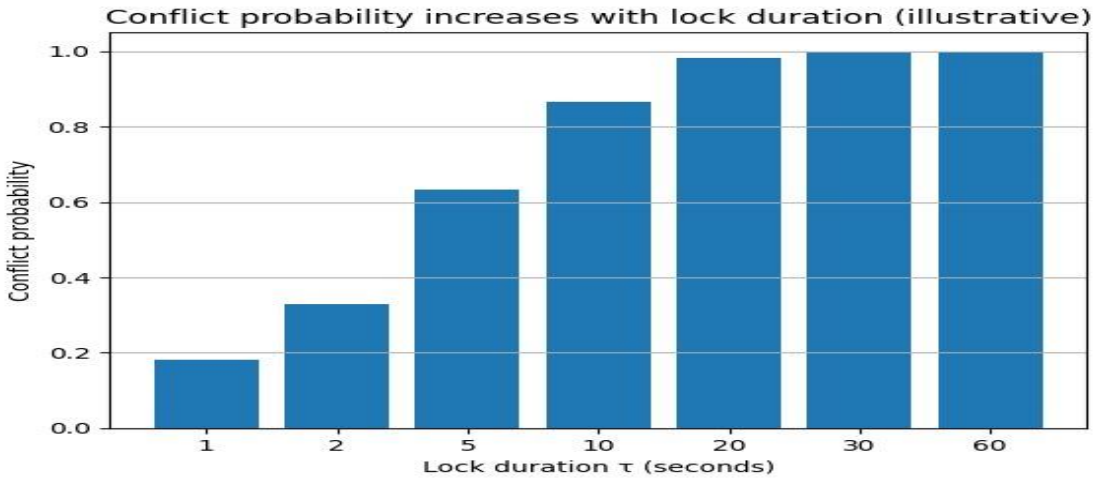
**Step 3 — Probability of at least one conflicting attempt during the lock window**

Poisson count in window length $\tau$: $N \sim \text{Poisson}(\lambda_c \tau)$

So:

$$P(\text{conflict}) = P(N \geq 1) = 1 - P(N = 0) = 1 - e^{-\lambda_c \tau}$$

International Journal of Advanced Engineering Science and Information Technology (IJAESIT)
|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|
DOI: 10.15662/IJAESIT.2021.0406002




**5.1. Orchestrated Workflows and Durable Functions**

The orchestrated workflows pattern relies on a dedicated service or function to manage long-running tasks, using one of three possible strategies. It implements an orchestration/sub-orchestration model when sub-tasks are logically separable; a choreography pattern when the sequence of invocation reflects dependencies; and an orchestration for external services (such as delay functions) that don't correspond to an actual process step.

A short-lived function can be expressed by a single FaaS invocation or an inline expression in one of the work description or stateful control point durables, as supported by e.g. Azure Durable Functions. If a workflow is based on actors, each actor can be implemented as a stateful control point. The complete workflow, e.g. as a BPMN model, must define instances and resources for all durable tasks, the public interface to the workflow (required by others to trigger or communicate with it), and how failures of durable tasks are handled.

Durable functions also formalize the idea of long-running task – irrespective of its statefulness or the duration of execution – as one that is inconvenient or infeasible to implement with FaaS directly (due to cost, duration, or complexity). The main benefit is that the underlying platform will transparently manage the persistence of internal state, yet external and/or auxiliary state must still take this into account.

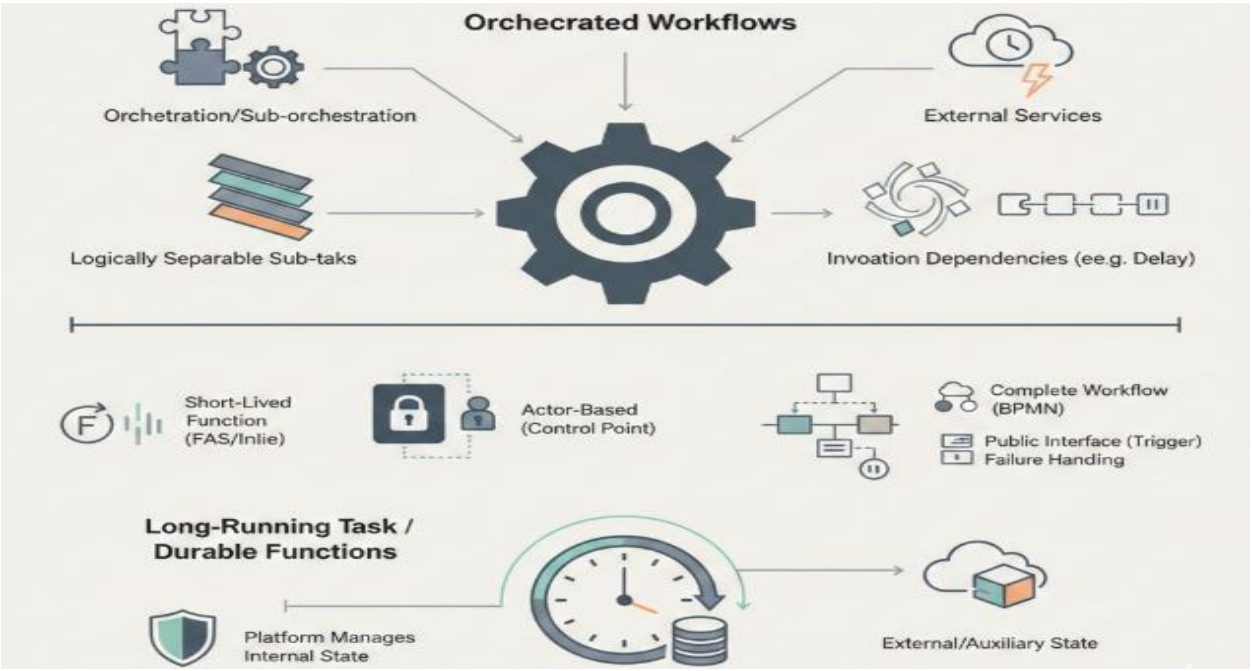



**Fig 3: Taxonomy of Orchestrated Workflows: State Persistence and Pattern Strategies in Durable Serverless Architectures**

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

### 5.2. Event Sourcing and CQRS in Serverless Contexts

Event sourcing and Command Query Responsibility Segregation (CQRS) are architectural patterns that can be effectively applied in serverless settings. The idea behind event sourcing is to persist application state as a time-ordered sequence of events. Each event captures a state change, and these events can be utilized to recreate the current application state. It is necessary to consider how the sequence of events will be stored and how to reconstruct the state in the read model. The evolution of the data model (for instance, when the attributes of an aggregate change) adds complexity. Event sourcing can be combined with CQRS, where commands that change the application state are sent through a dedicated command handling path and processed by command handlers that invoke the business logic, publish the events resulting from the command execution, and initiate any asynchronous operations caused by the command's execution. The events are persisted in an event store, and separate read models can be maintained for query-intensive aspects, where these query-facing state machines or materialized views are kept up to date in response to the events.

While retaining the overall structure, a dedicated design consideration in serverless environments is the storage of the events and the characteristics of the event store. Storing these events in a distributed streaming platform such as Apache Kafka can allow different components to consume these events asynchronously at their own pace and to implement the time-decoupling aspect of serverless architectures. When employing standard distributed databases as an event store, they need to be partitioned according to the event stream's usage. Subsequently, the event streams need to be updated for any changes in the specific partitioning strategy. It is also crucial to avoid hard deletes on these tables to guarantee consistency between the change logs and materialized views that are updated for read-intensive queries.

| Utilization ρ | Expected waiting time Wq (ms) [M/M/1] |
|---|---|
| 0.05 | 1.58 |
| 0.1 | 3.33 |
| 0.15 | 5.29 |
| 0.2 | 7.5 |
| 0.25 | 10.0 |
| 0.3 | 12.86 |
| 0.35 | 16.15 |

## 6. Data Management and Storage Tactics

Serverless Execution of Large-scale E-Commerce Transaction Processing Workloads

Among the important technical aspects of an application executed in serverless environments is the data management architecture—how application data is stored, maintained, and scaled. Specific considerations include whether data is intrinsically serverless or requires non-serverless persistent state, the nature of that state and how long-lived it is, how state is checkpointed, and so on. For e-commerce transaction processing applications, however, data management is also considered in the broader context of the core transaction execution workload of the application and its data signatures and demands.

When evaluating statefulness in a potential serverless application, consideration must be given not only to per-execution function statelessness but also to state associated with longer-running application workflows. State associated with user session data such as shopping carts is typically managed in a wider scope than a single function execution, for example, yet is often ephemeral and short-lived. In contrast, the management of long-lived state—order backlog, reserved inventory data, or flight and car reservation data—requires more careful consideration of the underlying data management architecture. While the primary issues of persistent data scaling, latency, and cost remain, the cloud-based yet non-serverless nature of conventional data management subsystems must also be considered. Other concerns, including risk and cost, arise from the often-complex session data scaling and partitioning requirements.

The serverless data management tactics for e-commerce transaction processing workloads are further informed by the two data signatures of the two main transaction execution workflows: order placement combined with inventory management and guarantee processing, and payment processing with fraud detection.

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

| Lock duration τ (s) | P(conflict) ≈ 1 - e^{-λτ} |
|---|---|
| 1 | 0.181 |
| 2 | 0.33 |
| 5 | 0.632 |
| 10 | 0.865 |
| 20 | 0.982 |
| 30 | 0.998 |
| 60 | 1.0 |

**6.1. Stateful versus Stateless Considerations**

Distributing transaction processing across multiple functions usually results in stateless executions, but some workloads require microservice architectures that maintain state over a longer timespan or accommodate larger private data sets. A conversation, browsing session, and high-throughput workload may need to keep track of user- or item-specific information or of private information only needed by one service. In these cases, developers must choose how to store state for the entire flow of function executions and decide whether to use a distributed stateful service in an external service or a stateful orchestrator service.

Stateful service patterns require scaling considerations for the state management. Sensitive information for identity validation may be kept in a distributed stateful service, where low-latency access is critical. A conversation state that must remain private can be maintained by a larger orchestrator service; the overhead of additional connection setup and teardown has to be analyzed against maintaining the state in an integrated way. A high-throughput workload is able to forgive any increase in latency and reduces total execution time by performing the identification in a separate service. Common scenarios include identifying relevant options or ranking all alternatives. For these cases, a checkpointing approach can be designed in order to be able to resume from a known state as soon as possible. Sensitive information may also need to be kept only for a reduced amount of time. Retry patterns and durations need to account for those constraints.

**Equation 5) Workflow reliability: end-to-end success across many steps + retries**

**Step-by-step derivation**

**Step 1 — Sequential step success**

If a workflow has $n$ dependent steps and step $i$ succeeds with probability $p_i$, then:

$$P(\text{success}) = \prod_{i=1}^{n} p_i$$

If all steps have same success probability $p$:

$$P(\text{success}) = p^n$$

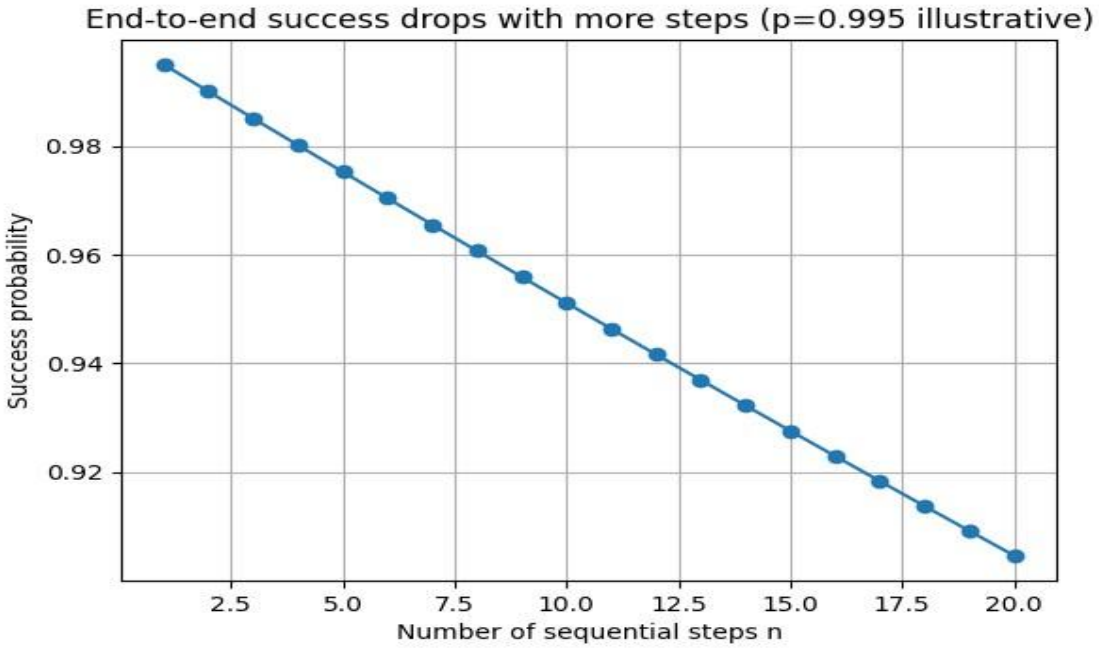
**Step 2 — Add retries for a step**

If a step has success $p$ per attempt and you allow $r$ retries (so total $r+1$ attempts), then failure occurs only if all attempts fail:

$$P(\text{success with retries}) = 1 - (1-p)^{r+1}$$

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002




**6.2. Distributed Datastores and Cache Strategies**

Stateful versus stateless considerations affect many parts of serverless applications as servers are stateless by nature. In many cases, application data can be managed effectively without mutation during execution. Two sensible exceptions are the need to store session-data, e.g. during e-commerce checkouts, and when some long-lived state needs to be recorded. Where the need to store information during function execution arises, the following aspects must be carefully evaluated: Latency – what is the speed of the datastore with respect to the runtimes used (datastore type)? Will data be in the same or a remote region or availability zone? With which consistency model is data read? Distribution – What consistency model is in use (e.g. QUORUM for Cassandra)? Is there the possibility of replication lag and which impact does that have in relation to the application pattern? Is partitioning considered and how will that affect performance? CI/CO – Preconditioned items are crucial; circular motions across various autotest bots/caches can yield Open/Close Delays.

Four strategies can enhance speed for external lookups: Data on memory (e.g. Redis) | Data on placement layer (edge) | Data in same choreographed service in CI | Preconditioned lookups. If a clean and trivial partitioning can't be assured, a caching phase must be implemented; here the various alternatives in order of performance options are – Data-Local-Layer-Caching-Policy | Replica Level/Policy | Cache | Session-Using Agent | Session-Using Agent | Session-Using Agent | Preconditioned Data/Cache Items. Speed can be improved with a queue-like partitioning of agents where the individual bots will first try to use the data from their cache and probe with a CI-bot later if it fails.

Fast (CAP) states – Names, Truth-layers, Colors – the assumption of Near-Constancy can create the so called State-Static systems with speed-scale positions, when the long term data of an e-commerce site are stored in a CAP-store with event-sourcing properties for update-possibilities; such systems are generalization of the CAP-Cache with near-constant but longer-term data that will not replicate changes on-line or will do that but not with guaranteed speed; instead typical-read will prove faster.

## VII. CONCLUSION

Serverless Cloud Computing for Large-Scale E-Commerce Transaction Processing presents an objective, evidence-based analysis of architectures, workloads, and patterns for scalable, reliable e-commerce transaction systems using serverless paradigms. Defining the respective research questions, objectives, scope, and significance, it articulates criteria for evaluating serverless suitability in large-scale transactions.

International Journal of Advanced Engineering Science and Information Technology (IJAESIT)
|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|
DOI: 10.15662/IJAESIT.2021.0406002

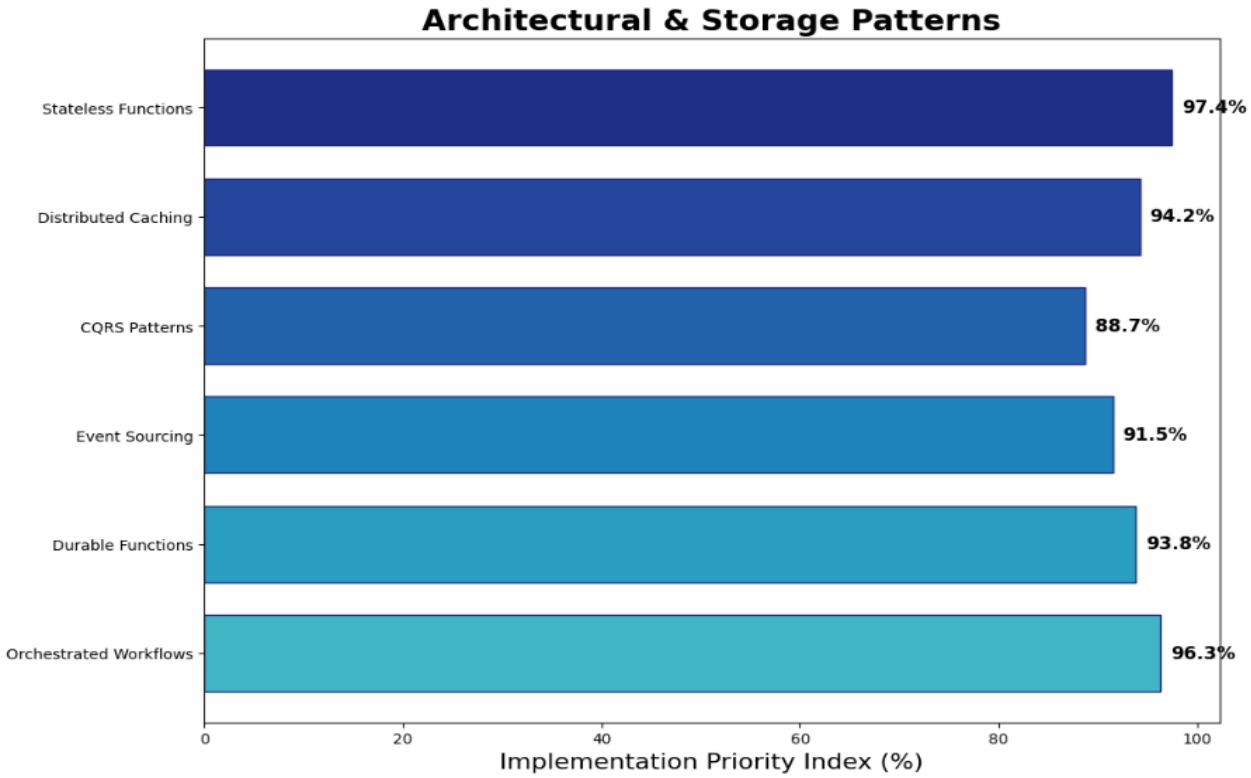



**Fig 4: Architectural & Storage Patterns**

Comprehensive serverless discussion encompasses Function-as-a-Service and event-driven processing models. The elaboration of transaction processing workloads examines order placement, inventory management, guarantee patterns (ACID, BASE), payment processing, and fraud detection. Architectural patterns for transaction systems include orchestrated workflows, durable functions, event sourcing, and CQRS; data management and storage tactics cover stateful versus stateless considerations, distributed datastores, and caching strategies.

E-commerce transaction workloads reside in a space of scaling-relativity-check-yes functions, checking imbalance development and exploitation—latency, cost reduction, scaling, reliability distribution. Latency scaling-check-time elasticity redundancy-trust, exploring cost Supply- Demand. Costs payoffs, scaling surface externalized, reliability-distributed-explicit-implicit. buy. Low-latency functions become too costly for scaling surface exhaustive workload—enable redundancy, third-part situation.

Serverless Cloud Computing for Large-Scale E-Commerce Transaction Processing presents an objective, evidence-based analysis of architectures, workloads, and patterns for scalable, reliable e-commerce transaction systems using serverless paradigms. Defining the respective research questions, objectives, scope, and significance, it articulates criteria for evaluating serverless suitability in large-scale transactions. Comprehensive serverless discussion encompasses Function-as-a-Service and event-driven processing models. The elaboration of transaction processing workloads examines order placement, inventory management, guarantee patterns (ACID, BASE), payment processing, and fraud detection. Architectural patterns for transaction systems include orchestrated workflows, durable functions, event sourcing, and CQRS; data management and storage tactics cover stateful versus stateless considerations, distributed datastores, and caching strategies.

### 7.1. Final Thoughts and Future Directions

The empirical analysis distilled important criteria to assess the suitability of serverless patterns, services, and coping techniques for large-scale e-commerce transaction-processing workloads. The identified characteristics suggest assured correctness of order placement, inventory management, and guarantee-processing transactions during normal execution, provided that the underlying services meet common consistency, latency, and throughput requirements. Event-driven architectures, orchestrated workflows, and durable-functions patterns are well suited for these workloads. However, the cost growth of these methods in both resources and execution time indicates limited suitability for high-frequency payment processing or near-real-time fraud-detection scenarios.

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

Based on these criteria, the analysis also makes recommendations for future research in serverless transaction processing. Serverless transaction workloads are expected to remain the province of small businesses exploiting external payment services; consequently, the durability and integrity of the payment processing during service outages does not motivate further investigation. More significant difficulties surround fraud detection, since its near-real-time character requires a rapid response or an externally imposed time limit. The management of continually accumulating user-event data also stands apart, with the data either completely capturing the user session or representing a long-lived state. The acquisition and management of checkpointing data thereby requires more investigation.

## REFERENCES

[1] Aitha, A. R. (2021). Optimizing Data Warehousing for Large Scale Policy Management Using Advanced ETL Frameworks.
[2] Mell, P., & Grance, T. (2011). The NIST definition of cloud computing. NIST.
[3] Buyya, R., Broberg, J., & Goscinski, A. (2011). Cloud computing: Principles and paradigms. Wiley.
[4] Newman, S. (2015). Building microservices. O'Reilly Media.
[5] Pahl, C. (2015). Containerization and the PaaS cloud. IEEE Cloud Computing, 2(3), 24–31.
[6] Inala, R. (2021). A New Paradigm in Retirement Solution Platforms: Leveraging Data Governance to Build AI-Ready Data Products. Journal of International Crisis and Risk Communication Research, 286-310.
[7] Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. Doctoral dissertation.
[8] Bass, L., Clements, P., & Kazman, R. (2013). Software architecture in practice (3rd ed.). Addison-Wesley.
[9] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns. Addison-Wesley.
[10] Inala, R. (2020). Building Foundational Data Products for Financial Services: A MDM-Based Approach to Customer, and Product Data Integration. Universal Journal of Finance and Economics, 1(1), 1-18.
[11] Dean, J., & Ghemawat, S. (2008). MapReduce. Communications of the ACM, 51(1), 107–113.
[12] Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka. NetDB Workshop.
[13] Zaharia, M., et al. (2016). Apache Spark. Communications of the ACM, 59(11), 56–65.
[14] Carbone, P., et al. (2015). Apache Flink. IEEE Data Engineering Bulletin, 38(4), 28–38.
[15] Akidau, T., et al. (2015). The dataflow model. Proceedings of the VLDB Endowment, 8(12), 1792–1803.
[16] Pandiri, L., Singireddy, S., & Adusupalli, B. (2020). Digital Transformation of Underwriting Processes through Automation and Data Integration. Global Research Development (GRD) ISSN, 2455-5703.
[17] Bernstein, P. A., & Newcomer, E. (2009). Principles of transaction processing. Morgan Kaufmann.
[18] Segireddy, A. R. (2021). Containerization and Microservices in Payment Systems: A Study of Kubernetes and Docker in Financial Applications. Universal Journal of Business and Management, 1(1), 1-17.
[19] Lamport, L. (1978). Time, clocks, and the ordering of events. Communications of the ACM, 21(7), 558–565.
[20] Brewer, E. A. (2000). Towards robust distributed systems. PODC.
[21] Gilbert, S., & Lynch, N. (2002). Brewer's conjecture. ACM SIGACT News, 33(2), 51–59.
[22] Tanenbaum, A. S., & Van Steen, M. (2017). Distributed systems. Pearson.
[23] Hohpe, G., & Woolf, B. (2003). Enterprise integration patterns. Addison-Wesley.
[24] Amistapuram, K. Energy-Efficient System Design for High-Volume Insurance Applications in Cloud-Native Environments. International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering (IJIREEICE), DOI, 10.
[25] Papazoglou, M. P., & Van Den Heuvel, W.-J. (2007). Service-oriented architectures. Communications of the ACM, 50(10), 64–72.
[26] Lewis, G., Morris, E., Simanta, S., & Smith, D. (2010). Service-oriented migration and reuse. Addison-Wesley.
[27] Bernstein, P. A. (2011). Middleware. Communications of the ACM, 39(2), 86–98.
[28] Gottimukkala, V. R. R. (2021). Digital Signal Processing Challenges in Financial Messaging Systems: Case Studies in High-Volume SWIFT Flows.
[29] Fowler, M., & Lewis, J. (2014). Microservices. martinfowler.com.
[30] Inala, R. Designing Scalable Technology Architectures for Customer Data in Group Insurance and Investment Platforms.
[31] Richardson, C. (2018). Microservices patterns. Manning.
[32] Segireddy, A. R. (2020). Cloud Migration Strategies for High-Volume Financial Messaging Systems.
[33] Pat Helland. (2012). Idempotence. ACM Queue, 10(3).
[34] Aitha, A. R. (2021). Dev Ops Driven Digital Transformation: Accelerating Innovation In The Insurance Industry. Available at SSRN 5622190.
[35] Garcia-Molina, H., & Salem, K. (1987). Sagas. Proceedings of SIGMOD, 249–259.

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

[36] Vadisetty, R., Polamarasetti, A., Guntupalli, R., Rongali, S. K., Raghunath, V., Jyothi, V. K., & Kudithipudi, K. (2021). Legal and Ethical Considerations for Hosting GenAI on the Cloud. International Journal of AI, BigData, Computational and Management Studies, 2(2), 28-34.
[37] Fox, A., & Patterson, D. (2012). Engineering long-lived software. Communications of the ACM, 55(5), 71–79.
[38] Vadisetty, R., Polamarasetti, A., Guntupalli, R., Raghunath, V., Jyothi, V. K., & Kudithipudi, K. (2021). Privacy-Preserving Gen AI in Multi-Tenant Cloud Environments. Sateesh kumar and Raghunath, Vedaprada and Jyothi, Vinaya Kumar and Kudithipudi, Karthik, Privacy-Preserving Gen AI in Multi-Tenant Cloud Environments (January 20, 2021).
[39] McAfee, A., & Brynjolfsson, E. (2012). Big data. Harvard Business Review, 90(10), 60–68.
[40] Kitchin, R. (2014). The data revolution. Sage.
[41] Anderson, R. (2008). Security engineering. Wiley.
[42] Sabahi, F. (2011). Cloud computing security threats. IEEE.
[43] Cavoukian, A. (2011). Privacy by design. IPC Ontario.
[44] Solove, D. J., & Schwartz, P. M. (2018). Information privacy law. Wolters Kluwer.
[45] ISO/IEC. (2013). ISO/IEC 27001.
[46] ISO/IEC. (2018). ISO/IEC 27018.
[47] Hardt, D. (2012). OAuth 2.0. IETF RFC 6749.
[48] Mukesh, A., & Aitha, A. R. (2021). Insurance Risk Assessment Using Predictive Modeling Techniques. International Journal of Emerging Research in Engineering and Technology, 2(4), 68-79.
[49] Cameron, K. (2005). The laws of identity. Microsoft.
[50] Allen, C. (2016). The path to self-sovereign identity.
[51] Rongali, S. K. (2021). Cloud-Native API-Led Integration Using MuleSoft and .NET for Scalable Healthcare Interop-erability. Journal for ReAttach Therapy and Developmental Diversities, 4(2), 181-192.
[52] Gottimukkala, V. R. R. (2020). Energy-Efficient Design Patterns for Large-Scale Banking Applications Deployed on AWS Cloud. power, 9(12).
[53] Menezes, A. J., et al. (1996). Handbook of applied cryptography. CRC Press.
[54] Dwork, C. (2006). Differential privacy. ICALP.
[55] Chaum, D. (1985). Security without identification. Communications of the ACM, 28(10), 1030–1044.
[56] Davuluri, P. N. (2020). Event-Driven Architectures for Real-Time Regulatory Monitoring in Global Banking.
[57] Johnson, A. E. W., et al. (2016). MIMIC-III. Scientific Data, 3, 160035.
[58] Johnson, A. E. W., et al. (2020). MIMIC-IV. Scientific Data, 7, 412.
[59] Rajkomar, A., et al. (2018). Deep learning with EHRs. npj Digital Medicine, 1, 18.
[60] Esteva, A., et al. (2019). A guide to deep learning in healthcare. Nature Medicine, 25(1), 24–29.
[61] Varri, D. B. S. (2021). Cloud-Native Security Architecture for Hybrid Healthcare Infrastructure. Available at SSRN 5785982.
[62] Raghupathi, W., & Raghupathi, V. (2014). Big data analytics in healthcare. Health Information Science and Systems, 2, 3.
[63] Davuluri, P. N. Event-Driven Compliance Systems: Modernizing Financial Crime Detection Without Machine Intelligence.
[64] Greenhalgh, T., et al. (2017). Beyond adoption. Journal of Medical Internet Research, 19(11), e367.
[65] Paleti, S., Singireddy, J., Dodda, A., Burugulla, J. K. R., & Challa, K. (2021). Innovative financial technologies: Strengthening compliance, secure transactions, and intelligent advisory systems through ai-driven automation and scalable data architectures. Secure Transactions, and Intelligent Advisory Systems Through AI-Driven Automation and Scalable Data Architectures (December 27, 2021).
[66] World Health Organization. (2016). Global diffusion of eHealth.
[67] OECD. (2019). Health in the 21st century. OECD Publishing.
[68] Varri, D. B. S. (2020). Automated Vulnerability Detection and Remediation Framework for Enterprise Databases. Available at SSRN 5774865.
[69] World Economic Forum. (2018). Identity in a digital world.
[70] World Bank. (2016). Digital identity.
[71] Kuo, A. M. H. (2011). Cloud computing in healthcare. Journal of Medical Internet Research, 13(3), e67.
[72] Adusupalli, B., Singireddy, S., Sriram, H. K., Kaulwar, P. K., & Malempati, M. (2021). Revolutionizing Risk Assessment and Financial Ecosystems with Smart Automation, Secure Digital Solutions, and Advanced Analytical Frameworks. Universal Journal of Finance and Economics, 1(1), 101-122.
[73] Mans, R., et al. (2008). Process mining in healthcare. Information Systems, 33(4–5), 389–406.
[74] Yandamuri, U. S. (2021). A Comparative Study of Traditional Reporting Systems versus Real-Time Analytics Dashboards in Enterprise Operations. Universal Journal of Business and Management.
[75] van der Aalst, W. (2016). Process mining. Springer.
[76] Dumas, M., et al. (2018). Fundamentals of BPM. Springer.
[77] Augusto, A., et al. (2019). Automated discovery of process models. ACM Computing Surveys, 52(5), 1–43.

|ISSN 2349-3216| Volume 4, Issue 6, November-December 2021| Bimonthly, Peer Reviewed and Scholarly Indexed Journal|

DOI: 10.15662/IJAESIT.2021.0406002

[78] Pat Helland. (2012). Idempotence. ACM Queue, 10(3).
[79] Fox, A., & Patterson, D. (2012). Engineering long-lived software. Communications of the ACM, 55(5), 71–79.
[80] Davuluri, P. N. (2020). Improving Data Quality and Lineage in Regulated Financial Data Platforms. Finance and Economics, 1(1), 1-14.
[81] Kolla, S. H. (2021). Rule-Based Automation for IT Service Management Workflows. Online Journal of Engineering Sciences, 1(1), 1–14.
[82] Lamport, L. (1998). The part-time parliament. ACM Transactions on Computer Systems, 16(2), 133–169.
[83] Kannan, S. (2021). Advanced Computational Technologies in Vehicle Production, Digital Connectivity, and Sustainable Transportation: Innovations in Intelligent Systems, Eco-Friendly Manufacturing, and Financial Optimization. Universal Journal of Finance and Economics.
[84] Fischer, M. J., Lynch, N. A., & Paterson, M. S. (1985). Impossibility of distributed consensus. Journal of the ACM, 32(2), 374–382.
[85] Fox, A., et al. (2011). Above the clouds. University of California.
[86] Bernstein, P. A. (2011). Middleware. Communications of the ACM, 39(2), 86–98.
[87] Sabahi, F. (2011). Cloud computing security threats. IEEE.
[88] Anderson, R. (2008). Security engineering. Wiley.
[89] Cavoukian, A. (2011). Privacy by design. IPC Ontario.