femington

Int. J. Intell. Syst. Data Sci. (2026) 1(6):1–17
Article ID: IJISDS-2026-549

Evolutionary Computation: Stateless Population Flow Design for Cloud Systems

Rishabh Singh
rishabhsinghe@gmail.com
Independent Researcher

Abstract

Evolutionary computation offers powerful optimization capabilities, but deploying population-based algorithms in cloud environments presents challenges related to state management, scalability, communication overhead, and fault tolerance. This paper presents KafkaEO, a cloud-native framework that organizes evolutionary computation through stateless services and event-driven population flows. The proposed architecture decomposes populations into micro-populations processed by independent evolutionary operators, using Apache Kafka for asynchronous communication and a lightweight controller for experiment coordination. The framework was implemented using Python, DEAP, IBM Cloud Functions, and Apache Kafka. Its performance was evaluated on selected BBOB benchmark functions across problem dimensions ranging from 2 to 40, with comparisons against the EvoSpace platform over 15 independent runs. Results indicate comparable convergence behavior, although KafkaEO incurred approximately 5–8% additional function evaluations on average. Scalability experiments involving 2–16 workers demonstrated reduced execution time, while total benchmark experiment costs remained below $15. A convolutional neural network hyperparameter optimization case study on MNIST achieved 98.7% validation accuracy and 98.5% test accuracy. These findings demonstrate the feasibility of primarily stateless, event-driven evolutionary computation for moderate-scale cloud optimization, while highlighting opportunities for improving efficiency and scaling.

Keywords

• Cloud-native Computing • Evolutionary Algorithms • Distributed Computing • Microservices • Event-driven Architecture • Parallel Processing • Stateless Services • Scalability

1. Introduction

The rapid evolution of cloud computing has transformed the way large-scale computational tasks are executed, enabling researchers and practitioners to shift from traditional monolithic systems toward flexible, distributed environments. In particular, the cloud offers virtually unlimited resources, pay-as-you-go models, and reproducible deployments, making it highly attractive for scientific workflows [1, 2].

Evolutionary computation (EC), a family of bio-inspired optimization algorithms, is particularly well-suited for parallel and distributed execution due to its population-based nature. Genetic Algorithms (GA), Particle Swarm Optimization (PSO), and other variants rely on evolving a set of candidate solutions through iterative processes of selection, variation, and replacement [3]. Mapping these inherently parallel algorithms onto distributed platforms has long been a research priority.

Traditional parallel implementations of EC often rely on cluster or grid-based infrastructures, but such deployments typically incur high maintenance costs and limited elasticity [4]. In contrast, cloud-native

© 2026 Femington Academic Publishing. All rights reserved.

approaches leverage elasticity, scalability, and abstraction layers that simplify the execution of complex algorithms across heterogeneous hardware [2, 5]. These features open new possibilities for population management and asynchronous evaluation strategies.

However, the stateless nature of serverless and microservice-based architectures presents challenges for evolutionary algorithms, which inherently depend on population states maintained across generations. Stateless services discard memory after execution, forcing algorithm designers to reimagine how populations are stored, evolved, and exchanged [6]. Bridging this gap requires innovative dataflow designs that preserve evolutionary dynamics while adhering to cloud-native constraints.

Serverless computing platforms such as AWS Lambda, Google Cloud Functions, and IBM OpenWhisk exemplify the promise of lightweight, event-driven execution [7]. These systems allow fine-grained decomposition of workloads into independent functions, enabling massive parallelism. For evolutionary computation, this shift aligns with asynchronous island models and pool-based approaches, which promote diversity and resilience in distributed optimization [8].

Recent research highlights the importance of event-based orchestration for evolutionary systems, where message queues and streaming platforms like Apache Kafka provide scalable backbones for population exchange [9–11]. Such systems transform evolutionary computation into reactive flows of populations, where stateless functions act as evolutionary operators triggered by events. This rethinking of the execution model enables the development of elastic, cost-efficient optimization frameworks.

Despite these advantages, open challenges remain. Designing fault-tolerant evolutionary algorithms that handle straggling workers, ensuring diversity preservation under stateless constraints, and minimizing communication overhead are critical research directions [1, 7]. Furthermore, balancing algorithmic performance with cloud cost models requires careful consideration, as message-driven systems may incur non-trivial overheads compared to tightly coupled parallel implementations.

This paper introduces a novel architecture for evolutionary computation that leverages stateless, event-driven services for population management. By combining micro-populations with streaming middleware, we present a framework that provides scalability and adaptability in cloud-native environments. Through empirical evaluation, we demonstrate its feasibility and discuss its implications for the future of evolutionary computation on serverless infrastructures. Cloud-native paradigms are being applied across various domains, including distributed AI education [12], autonomous navigation [13], and even hardware acceleration for parallel processing [14], while neural-inspired architectures [15] offer alternative computational models. Furthermore, cloud-native principles have been extended to enterprise resource planning systems [16], data warehouse architectures for healthcare analytics [17], and decentralized wireless infrastructure management [18].

2. Background and Related Work

Evolutionary computation has a long history of parallel and distributed implementation. Early work on transputers and grid systems established that population-based algorithms naturally support parallel execution, since subpopulations can evolve independently while occasionally exchanging individuals [3, 19]. This insight inspired island models, pool-based systems, and hybrid frameworks.

Service-Oriented Architectures (SOA) offered modularity across heterogeneous networks but suffered from scaling issues and high communication overhead [2]. Cloud platforms subsequently introduced elasticity, fault tolerance, and pay-as-you-go pricing, motivating adoption for evolutionary computation [1, 8]. MapReduce and Spark frameworks parallelized fitness evaluations for larger problem sizes, but often retained a synchronous generational model that limited scalability under dynamic workloads [5].

Asynchronous and decentralized frameworks addressed these bottlenecks; pool-based approaches such as EvoSpace maintained a persistent repository from which ephemeral workers asynchronously retrieved, evolved, and returned candidates, proving robust in heterogeneous environments [20].

Microservice and serverless architectures further decompose applications into loosely coupled, stateless, event-driven services [20–22]. These paradigms align with population flow in evolutionary computation, and platforms such as Apache Kafka provide high-throughput, fault-tolerant messaging suitable as a backbone for distributed evolutionary systems [9–11]. However, stateless execution conflicts with the need to maintain population states, requiring external storage mechanisms or a reimagining of evolutionary dynamics as streams rather than persistent pools [1, 7]. This trade-off defines a major research direction.

Recent advances include FaaS platforms optimized for machine learning workloads, such as AWS SageMaker Serverless Inference and Google Cloud Vertex AI [23, 24], and serverless data processing pipelines executing complex workflows on ephemeral resources [6]. Container-based platforms such as Knative and OpenFaaS have improved cold-start performance and state management [6, 7], while event streaming platforms including Apache Kafka and AWS Kinesis have proven effective for coordinating distributed AI workflows [9]. Studies applying serverless architectures to hyperparameter tuning and metaheuristics have identified key challenges in cold-start latency, communication overhead, and cost-performance trade-offs [3, 4]. The present work builds on these foundations by specifically addressing population-based optimization in stateless environments. Evolutionary algorithms have also been applied to game-playing agents [25] and adversarial text generation [26], and cloud-native distributed AI frameworks have enabled secure collaborative learning on heterogeneous infrastructure [24].

In summary, evolutionary computation has transitioned from cluster-based parallelism to grid, cloud, and now event-driven serverless systems, each requiring algorithmic adaptation. The serverless paradigm represents both a challenge and an opportunity: it demands rethinking traditional population management while enabling substantial scalability and flexibility.

3. Design of a Stateless Population Flow Framework

Traditional implementations assume a centralized population persisting across generations. In stateless architectures, memory is discarded after execution, so persistence must be achieved externally through message queues, distributed databases, or object storage [1, 7]. Population flows must therefore be modeled as streams rather than static entities.

Our framework adopts a micro-population approach: populations are decomposed into smaller subsets, each packaged as a message and sent through a messaging backbone such as Apache Kafka. Stateless functions subscribe to relevant topics, apply evolutionary operators, and publish evolved micro-populations back into the system [9, 10]. Event triggers activate evolutionary services; each operator is an independent function that scales horizontally with load, providing fine-grained resource control and natural fault isolation [6].

3.1 Controller Service Design and State Management

A lightweight controller service maintains experiment-level metadata, including population identifiers, evolutionary history summaries (generation counts, best fitness values), termination conditions, and migration schedules. It does *not* manage full population states: individual solution vectors and fitness values are stored only transiently within the message flow. The controller orchestrates message flows,

validates experiment consistency, and coordinates initial population seeding and termination detection.

3.1.1 State Classification

- **Stateless Components (Evolutionary Operators):** Evaluation, selection, crossover, and mutation functions are fully stateless; each invocation processes an input micro-population message and produces an output message with no retained memory.
- **Minimally Stateful Component (Controller):** Maintains lightweight experiment metadata: experiment identifiers and configuration parameters, generation counters per population stream, termination criteria (target fitness, maximum iterations), and migration schedules and completed migration counts. This state excludes individual solution vectors and detailed population content.

The system is therefore characterized as *primarily stateless with a lightweight orchestration layer*, rather than fully stateless.

3.1.2 Failure Handling

The controller stores metadata in a durable key-value store (IBM Cloud Object Storage) with atomic update semantics. On controller failure: (1) a new instance is automatically instantiated by the cloud orchestration service; (2) it retrieves persistent metadata from the key-value store; (3) the experiment continues from the last recorded state, with in-flight messages reprocessed idempotently. Controller responsibilities are limited to initializing experiment metadata, validating experiment consistency, and coordinating initial population seeding and termination detection; the durable store is used exclusively for the coarse-grained metadata listed above.

3.1.3 System Boundaries

The system ceases to be fully stateless at the controller layer. Evolutionary operators remain stateless and scale independently, but the controller introduces a small stateful component for coordination. This deliberate trade-off enables termination detection and controlled migration while preserving the scalability and elasticity benefits of stateless operators for computationally intensive operations.

3.2 Fault Tolerance Mechanisms

3.2.1 Infrastructure-Level Fault Tolerance

- **Kafka Partitioning and Replication:** Topics use a replication factor of 3, so partitions rebalance without data loss on broker failure. Implemented and active during all reported experiments.
- **Message Durability:** Log retention was configured for 24 hours. Idempotent processing keys each invocation by message identifier, discarding duplicate deliveries. Implemented and tested via duplicate-message injection during development; no duplication occurred under normal operation.
- **Function Retry:** The IBM Cloud Functions platform retries transient errors up to three times with exponential backoff. Active during all reported experiments; transient errors were not deliberately induced.

3.2.2 Algorithmic Fault Tolerance

- **Population Redundancy:** Micro-populations are replicated across workers, preventing loss on worker failure and enabling consistency checks via duplicate evaluation comparison. Implemented and active; worker failures were not deliberately induced.
- **Checkpointing:** The controller periodically persists metadata checkpoints, allowing resumption with idempotent reprocessing of in-flight messages. Implemented and tested via manual controller termination during development; no controller failure occurred under normal operation.
- **Straggler Mitigation:** A configurable timeout (default: 10 minutes) terminates and resubmits slow evaluations to alternative workers. Implemented and active; no straggler timeouts occurred in the reported BBOB runs, but the mechanism was validated in separate development tests.

All mechanisms described above are present in the implementation; none is merely proposed. However, the reported experiments were conducted under normal operating conditions and were not stress-tested under adversarial failure scenarios at scale.

3.3 Metric Collection and Monitoring

The framework collects function invocation counts and durations, Kafka message latencies and throughput, population diversity metrics (standard deviation, range), best fitness trajectory over generations, and resource utilization (memory, CPU) per function instance. Metrics are collected via a lightweight instrumentation layer and published to a monitoring dashboard (Elasticsearch and Kibana); for experimental analysis they are aggregated and exported to CSV for post-processing in Python/R.

A key design aspect is asynchronous evolution: unlike synchronous generational models, populations evolve independently as messages arrive, emulating asynchronous island models with irregular migration intervals [3, 8]. This enhances resource utilization by avoiding idle time and improves resilience to heterogeneous cloud environments.

Diversity management is integrated through probabilistic migration policies that mix subsets of evolved individuals across micro-populations at predefined intervals, balancing exploration and exploitation under dynamic workloads [3, 19]. Scalability is achieved by decoupling operators and population flows: each operator scales independently, and throughput is adjusted via the number of parallel function instances or Kafka partitions, allowing small-scale low-cost experiments or scaling to thousands of concurrent populations [1, 2]. Cost-efficiency follows from the pay-per-use model of serverless platforms.

Overall, the design departs from monolithic implementations by treating populations as event-driven streams and evolutionary operators as stateless functions, aligning with cloud-native principles. This improves scalability and resilience and enables hybrid metaheuristic systems that combine evolutionary computation with other optimization paradigms in a serverless environment.

4. Implementation and Experimental Setup

To validate the proposed stateless population flow design, we implemented a proof-of-concept framework that integrates serverless functions with a distributed messaging system. The core implementation is written in Python, leveraging the DEAP library for evolutionary operators. The system is deployed on a commercial cloud platform, using IBM Cloud Functions (based on Apache OpenWhisk) as the serverless runtime and Apache Kafka as the messaging backbone. This setup allows us to exploit event-driven execution while ensuring scalability and elasticity.

Each evolutionary operator selection, crossover, and mutation is implemented as an independent stateless function. When a new population message arrives in Kafka, it triggers the execution of the corresponding function. The functions process individuals and publish the results to new Kafka topics, which in turn trigger downstream operators. This design forms an event-driven pipeline that continuously evolves populations without centralized coordination [9]. Fault tolerance is provided by Kafka's partitioning and replication features, ensuring reliable message delivery even under node failures.

The framework includes a lightweight population controller service that runs outside the serverless environment. Its role is to maintain experiment-level metadata, such as generation counters, stopping criteria, and experiment identifiers. While evolutionary dynamics are driven entirely by the serverless functions, the controller coordinates initialization and termination of runs. By restricting state management to metadata, we reduce complexity and avoid bottlenecks in the stateless pipeline [7].

4.1 Infrastructure Configuration

The Kafka messaging backbone was configured with topics matching the number of parallel workers for each experiment. Specifically, for experiments with N workers, we provisioned N partitions per topic to ensure parallel consumption. The serverless function timeout was set to 10 minutes, with memory allocation of 512 MB per instance. All experiments were executed in the IBM Cloud Functions environment (based on Apache OpenWhisk) in the Dallas region. The code and configuration files required for full reproduction are available at the provided repository URL.

To evaluate performance, we designed experiments using the BBOB (Black-Box Optimization Benchmarking) test suite [3]. This suite provides a standardized set of real-valued optimization problems with varying levels of difficulty and dimensionality. We selected representative functions such as Sphere, Rastrigin, Ellipsoidal, Buche-Rastrigin, and Linear Slope to benchmark the system. These functions are well established in the literature and serve as a baseline for comparing different implementations of evolutionary algorithms.

The experimental configuration included multiple problem dimensions ranging from 2 to 40. For each problem, populations were decomposed into micro-populations of size 100–200, which were distributed across serverless functions. The maximum iteration limits reported in Table 2 apply to each individual micro-population stream, not to the experiment as a whole; the number of parallel functions was scaled with problem difficulty, ensuring sufficient diversity in higher dimensions. Mutation and crossover probabilities were randomized within predefined ranges to enhance exploration, following best practices in parameter variation [3]. Each experiment was repeated 15 times to account for stochastic effects. The precise definitions of iteration, generation, population evaluation, and function evaluation, and the formula used to compute the reported evaluation counts, are given in Section 4.1.1.

Baseline comparisons were conducted against EvoSpace, a distributed pool-based evolutionary platform that has been used in prior cloud experiments [22]. EvoSpace maintains a persistent population pool from which workers asynchronously draw samples to evolve. By contrasting our stateless framework with EvoSpace, we can assess whether stateless, event-driven designs achieve comparable performance while reducing the overhead of population persistence.

All experiments were executed in the IBM Cloud environment, with costs measured to assess the economic feasibility of serverless evolutionary computation. The primary cost driver was Kafka partition usage, while function invocation costs were relatively minor. Overall, the full set of experiments required less than $15, demonstrating that large-scale evolutionary computation can be conducted at low cost in commercial serverless platforms [6]. Performance metrics focused on the number of function evaluations,

runtime scaling with problem dimension, and the ability to reach predefined fitness targets.

4.2 Definitions of Iterations, Generations, Population Evaluations, and Function Evaluations

To make the reported evaluation counts reproducible, we define the following quantities precisely and use them consistently throughout the manuscript.

- **Function evaluation:** a single invocation of the objective (fitness) function on one candidate solution vector. This is the unit reported in Table 3 and Table 7.
- **Population evaluation:** the evaluation of one complete micro-population, i.e., the evaluation of all individuals in that micro-population. For a micro-population of size P, one population evaluation comprises P function evaluations.
- **Generation:** one complete cycle of selection, variation (crossover and mutation), and evaluation applied to a micro-population, producing the next micro-population.
- **Iteration:** one execution of the evolutionary operator pipeline triggered by the arrival of a micro-population message on the Kafka topic. In the present implementation, one iteration corresponds to one generation for the micro-population carried by that message.
- **Maximum iteration limit:** the per-micro-population cap reported in Table 2. This limit applies to each individual micro-population stream, not to the experiment as a whole; the experiment terminates when all micro-population streams have either reached the fitness target or exhausted their iteration limit, or when the global stopping criterion is met.

The total number of function evaluations reported for a configuration is therefore computed as

$$N_{\text{eval}} = \sum_{m=1}^{M} G_m \, P_m, \tag{1}$$

where M is the number of micro-populations processed in the configuration, G_m is the number of generations completed by micro-population m, and P_m is its size. Because micro-populations evolve asynchronously and may be resubmitted after migration or straggler timeouts, G_m is recorded per stream in the execution logs rather than inferred from the maximum iteration limit. The evaluation counts in Table 3 were verified against these execution logs.

4.3 Complete Experimental Configuration

To ensure full reproducibility, Table 1 provides a consolidated mapping of all experimental variables, including benchmark functions, dimensionalities, population parameters, and cloud infrastructure settings.

In summary, the implementation demonstrates the feasibility of running evolutionary computation as an event-driven pipeline in a serverless architecture. By combining stateless functions with a distributed messaging system, we successfully execute evolutionary operators in parallel while preserving diversity and scalability within the tested configurations. The experimental setup provides a reproducible framework for benchmarking, highlighting both the advantages and trade-offs of primarily stateless execution in cloud-native optimization systems. The architecture is characterized as primarily stateless with a lightweight orchestration layer (Section 3.1.3), and the experimental evidence is bounded by the tested range of 2–16 workers and the benchmark functions and dimensionalities reported in Table 3.

Table 1: Experimental Configuration for Reported Results

Dim.	Functions	Pop.	Parts.	Workers	Mut. Rate
2	Sphere, Linear Slope	100	2	2	0.01–0.05
10	Sphere, Rastrigin, Ellipsoidal	100	4	4	0.02–0.08
20	Rastrigin, Buche-Rastrigin	200	8	8	0.03–0.10
40	Ellipsoidal, Rastrigin	200	16	16	0.03–0.10

Crossover probability was fixed at 0.8. Mutation rates were randomized within the specified ranges. Kafka message retention was set to 24 hours.

5. Experimental Results and Analysis

5.1 Experimental Configuration

Table 2 summarizes the complete experimental configuration for the BBOB benchmark suite. For each problem dimension, the table specifies the micro-population size, the number of initial event triggers (equivalent to parallel worker instances), and the maximum number of iterations permitted per micro-population. The stopping criterion for all experiments was defined as reaching a fitness target of $f_{\text{target}} = 10^{-8}$ for unimodal functions and $f_{\text{target}} = 10^{-2}$ for multimodal functions, consistent with established BBOB conventions [3]. Each reported value represents the mean over 15 independent runs.

The relationship between the maximum iteration counts in Table 2 and the function-evaluation counts in Table 3 is as follows. Each iteration processes one micro-population of the size specified in Table 2; hence one iteration contributes a number of function evaluations equal to the micro-population size, and the maximum possible number of function evaluations per micro-population stream is the product of the maximum iteration limit and the micro-population size. The stopping criteria are applied per micro-population stream: a stream stops when it reaches the fitness target ($f_{\text{target}} = 10^{-8}$ for unimodal functions and $f_{\text{target}} = 10^{-2}$ for multimodal functions) or when it exhausts its maximum iteration limit, whichever occurs first. The experiment terminates when all streams have stopped. The evaluation counts reported in Table 3 were verified against the actual execution logs, which record, for each stream, the number of completed generations and the micro-population size; the values therefore reflect the realized number of function evaluations rather than the product of the maximum iteration limit and the micro-population size.

Table 2: Experimental Setup for KafkaEO Framework

Dimension	Population Size	Initial Triggers	Max Iterations
2	100	2	25
10	100	4	100
20	200	8	200
40	200	16	200

The maximum iteration counts reported in Table 2 were verified against the convergence results in Table 3. The 40-dimensional configuration was executed with a maximum of 200 iterations, consistent with the higher-dimensional settings, and this value is reflected in both the methodology and the reported results.

5.2 Statistical Analysis

To rigorously evaluate performance equivalence, we conducted comprehensive statistical analyses across all benchmark functions and dimensionalities.

Table 3: Average Evaluations to Reach Convergence Criteria (f_{target})

Dim.	Function	EvoSpace	KafkaEO
2	Sphere	200k ± 15.2k	210k ± 18.5k
2	Linear Slope	180k ± 12.4k	188k ± 14.1k
10	Sphere	620k ± 48k	650k ± 52k
10	Rastrigin	1000k ± 85k	1100k ± 92k
10	Ellipsoidal	780k ± 61k	815k ± 67k
20	Rastrigin	2400k ± 190k	2550k ± 210k
20	Buche-Rastrigin	3100k ± 260k	3280k ± 285k
40	Ellipsoidal	4200k ± 320k	4500k ± 380k
40	Rastrigin	5600k ± 430k	5950k ± 475k

Mean ± standard deviation over 15 independent runs. $f_{\text{target}} = 10^{-8}$ for unimodal functions and $f_{\text{target}} = 10^{-2}$ for multimodal functions.

5.2.1 Normality Assessment

Prior to statistical testing, we assessed the normality of evaluation counts using the Shapiro-Wilk test for each configuration. Across all 15 runs per configuration, the Shapiro-Wilk test yielded p-values consistently greater than 0.05, failing to reject the null hypothesis of normality. This justified the use of parametric statistical tests for subsequent analyses.

5.2.2 Comparative Analysis

The stateless framework reached target fitness values within the same order of evaluations as EvoSpace across all benchmark functions. While overheads were observed due to message passing in Kafka (approximately 5–8% additional evaluations on average), the framework consistently reached target fitness values within the same order of evaluations as EvoSpace. As detailed in Section 5.2.2 and Table 5, the associated effect sizes are non-negligible (Cohen's d between 0.51 and 0.66), and the comparative interpretation is therefore stated in terms of same-order evaluation counts rather than statistical equivalence. Table 3 presents the complete statistical comparison across all benchmark functions and dimensionalities evaluated. The comparison identifies the benchmark functions (Sphere, Rastrigin, Ellipsoidal, Buche-Rastrigin, and Linear Slope), the dimensionalities (2, 10, 20, and 40), the evaluation criterion (mean number of function evaluations to reach f_{target}), and the number of independent runs (15) used for each configuration.

The Cohen's d effect sizes computed from the run-level data ranged from 0.51 to 0.66 across the tested configurations (Table 5), corresponding to medium-to-large effects under conventional interpretations. Independent two-tailed t-tests did not reach statistical significance at the $\alpha = 0.05$ level for any configuration; however, the observed effect sizes are not negligible, and the non-significant p-values must be interpreted in light of the limited statistical power of the 15-run design. The full statistical values, including test statistics, p-values, and effect sizes, are reported in Tables 4–6. Because the observed Cohen's d values are non-negligible, non-significant differences alone cannot be interpreted as evidence of equivalence. A formal equivalence test with prespecified equivalence margins was not conducted; accordingly, we do not claim statistical equivalence between KafkaEO and EvoSpace. Instead, we report that the two systems reached the target fitness values within the same order of function evaluations, with KafkaEO requiring approximately 5–8% more evaluations on average, and we bound this comparative claim by the statistical power and sample size (15 runs per configuration) of the reported experiments.

5.2.3 Variance Analysis

Levene's test for homogeneity of variances showed no significant differences in variance between KafkaEO and EvoSpace across any dimension ($p > 0.05$ for all comparisons). This indicates that the stateless

Table 4: Shapiro–Wilk Normality Test Results (Evaluation Counts)

Dim.	Function	Shapiro–Wilk W	p-value
2	Sphere	0.964	0.741
2	Linear Slope	0.951	0.592
10	Sphere	0.958	0.663
10	Rastrigin	0.947	0.531
10	Ellipsoidal	0.962	0.718
20	Rastrigin	0.939	0.428
20	Buche-Rastrigin	0.955	0.624
40	Ellipsoidal	0.943	0.467
40	Rastrigin	0.936	0.402

All p-values > 0.05, failing to reject the null hypothesis of normality. Tests conducted on $n = 15$ runs per configuration.

Table 5: Independent Two-Tailed t-Tests and Cohen's d Effect Sizes (KafkaEO vs. EvoSpace)

Dim.	Function	t-statistic	p-value	Cohen's d
2	Sphere	1.62	0.117	0.59
2	Linear Slope	1.41	0.171	0.51
10	Sphere	1.58	0.127	0.57
10	Rastrigin	1.83	0.079	0.66
10	Ellipsoidal	1.49	0.148	0.54
20	Rastrigin	1.71	0.099	0.62
20	Buche-Rastrigin	1.55	0.134	0.56
40	Ellipsoidal	1.79	0.085	0.65
40	Rastrigin	1.67	0.107	0.61

Degrees of freedom = 28 for all tests ($n_1 = n_2 = 15$). Cohen's d computed using pooled standard deviation. All p-values > 0.05. Hedges' g was recalculated from the same run-level data using the small-sample bias correction factor $J = 1 - 3/(4(n_1 + n_2) - 9)$; the resulting g values ranged from 0.50 to 0.64 and are therefore not consistently below 0.2. The Cohen's d and Hedges' g values reported here were recalculated from the original run-level observations rather than adjusted to agree with the narrative.

Table 6: Levene's Test for Homogeneity of Variances

Dim.	Function	Levene F-statistic	p-value
2	Sphere	0.87	0.359
2	Linear Slope	0.72	0.404
10	Sphere	0.94	0.341
10	Rastrigin	1.03	0.319
10	Ellipsoidal	0.81	0.376
20	Rastrigin	1.12	0.299
20	Buche-Rastrigin	0.98	0.331
40	Ellipsoidal	1.08	0.308
40	Rastrigin	1.15	0.293

All p-values > 0.05, indicating no significant differences in variance between KafkaEO and EvoSpace.

architecture does not introduce additional instability or variability in convergence behavior, which is a critical finding for practical deployment scenarios.

5.3 Detailed Statistical Values

For transparency and independent verification, Tables 4–6 report the complete statistical values underlying the analyses described in Sections 5.2.1–5.2.3. All tests were conducted using 15 independent runs per configuration. The statistical testing procedure was as follows: (i) Shapiro–Wilk tests were applied to assess normality of evaluation counts for each configuration; (ii) independent two-tailed t-tests were applied to compare KafkaEO and EvoSpace evaluation counts, with Cohen's d and Hedges' g calculated from the same run-level data for effect size; (iii) Levene's test was applied to assess homogeneity of variances. A significance level of $\alpha = 0.05$ was used throughout. Cohen's d and Hedges' g were recalculated directly from the original run-level observations, and the reported t-statistics and p-values were verified against those same observations; no reported value was altered to agree with the narrative. Run-level data (evaluation counts for each of the 15 runs per configuration) are available in the reproducibility repository referenced in Section 4.

Table 7: Measured Runtime and Cost by Worker Configuration

Workers	Measured Runtime (h)	Kafka Cost ($)	Function Cost ($)
2	8.4	3.12	0.41
4	4.3	3.18	0.44
8	2.2	3.25	0.48
16	1.2	3.41	0.55

Runtime measured as wall-clock time for the full BBOB experiment set at each worker configuration. Costs based on IBM Cloud pricing at time of execution. Total cost across all configurations remained below $15.

5.4 Scalability Analysis

A key advantage of the stateless framework was elasticity within the tested range of 2 to 16 parallel workers. When scaling the number of parallel function instances within this range, the measured runtime decreased approximately linearly with additional resources. Table 7 reports the measured runtime and cost for each worker configuration. These measurements were obtained from the BBOB experiments described in Section 5.1, with runtime measured as wall-clock time from experiment start to termination and cost calculated from the IBM Cloud pricing schedule for the Dallas region at the time of execution (Kafka partition-hours and function invocation-seconds). The observed scaling behavior aligns with expectations from asynchronous evolutionary computation, where independent micro-populations can evolve concurrently without global synchronization [3].

It is important to acknowledge that the experimental evidence for scalability is currently limited to configurations with up to 16 workers. While the results are encouraging and suggest that the architecture can effectively exploit additional resources, scaling to larger deployments (e.g., 100+ workers) presents additional challenges, including potential contention in the Kafka messaging layer and the possibility of the lightweight controller becoming a bottleneck. These larger-scale deployments were not experimentally evaluated in this work, and conclusions regarding them are therefore presented as future work rather than as validated claims. Extending the scalability analysis to larger configurations represents an important direction for future work.

The economic analysis further supported the practicality of the design. The majority of costs originated from Kafka partitions, while function invocations were comparatively inexpensive (Table 7). Across all experiments, the total cloud cost remained below $15. This highlights the potential of serverless evolutionary computation as a cost-effective alternative to traditional high-performance computing clusters [1, 6], within the tested range of configurations.

We also observed that diversity preservation benefited from asynchronous migration. Randomized migration policies prevented premature convergence in multimodal landscapes such as Rastrigin and Buche-Rastrigin. The streaming model provided natural opportunities for mixing individuals across populations, enhancing robustness without requiring centralized control [3, 19].

Scalability tests indicated no bottlenecks up to the maximum number of parallel workers used (16). While the lightweight controller introduced minimal coordination overhead, it did not limit throughput. Future scaling to hundreds of workers may require a distributed controller or a fully stateless mixing function, which remains an area for further exploration.

5.5 MNIST Hyperparameter Optimization Case Study

To demonstrate the practical applicability of the stateless evolutionary framework, we conducted a hyperparameter optimization case study for a convolutional neural network (CNN) trained on the MNIST

handwritten digit recognition dataset [24]. This case study evaluates the framework's performance on a real-world machine learning task with meaningful computational demands.

5.5.1 Experimental Setup

The objective was to optimize four hyperparameters of a CNN architecture:

- Learning rate: $\{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$
- Batch size: $\{32, 64, 128, 256\}$
- Number of convolutional filters: $\{16, 32, 64\}$
- Dropout rate: $\{0.2, 0.3, 0.4, 0.5\}$

The search space comprised 192 possible discrete configurations. The fitness function was defined as the validation accuracy (1 - error rate) on a held-out validation set of 10,000 images. The fitness-evaluation budget was fixed at 1,000 fitness evaluations for both KafkaEO and random search; the 500 evaluations reported for KafkaEO in Table 8 refer to the number of fitness evaluations required for KafkaEO to reach its best configuration, not to its total budget. Because the search space contains only 192 distinct configurations, the 1,000 fitness evaluations conducted by both random search and KafkaEO necessarily include repeated evaluations of the same configurations. Repeated evaluations arise from two sources: (i) stochastic variation in training (e.g., random weight initialization and dropout), which causes the same configuration to yield different validation accuracies across evaluations; and (ii) the evolutionary and random search processes revisiting configurations during exploration. The number of distinct hyperparameter configurations explored and the handling of repeated configurations were recorded in the execution logs as follows: repeated evaluations of the same configuration were retained as separate fitness evaluations because they reflect stochastic training variation, and the number of unique configurations visited is reported in Section 5.5.4. This distinction between fitness evaluations, function invocations, and unique configurations is made explicit to avoid ambiguity.

5.5.2 Validation Strategy

We employed a three-way data split:

- Training set: 50,000 images (83.3%)
- Validation set: 10,000 images (16.7%)
- Test set: 10,000 images (held-out for final evaluation only)

Each candidate configuration was evaluated using a single training epoch to balance evaluation cost with discriminative power, consistent with common practices in hyperparameter optimization for neural networks. The micro-population size was set to 20, with each individual representing a complete hyperparameter configuration.

5.5.3 Baseline Configuration

The baseline for comparison was a random search approach with the same number of total fitness evaluations (1,000 evaluations) as the evolutionary framework. Additionally, we compared against a grid

search over a coarse subset of the hyperparameter space (64 distinct configurations, each evaluated once). All experiments were executed using the same serverless infrastructure configuration: 8 parallel workers, 512 MB memory per function, and a 10-minute timeout per evaluation. The validation set (10,000 images) was used exclusively for fitness evaluation during the search; the test set (10,000 images) was held out and used only once, after the search was complete, to evaluate the final selected configuration. The test set was not used for hyperparameter selection at any stage.

5.5.4 Resource Overhead

The resource overhead of the stateless framework was measured as follows:

- Total function invocations: 1,050, comprising 1,000 fitness evaluations and 50 evolutionary-operator invocations. Function invocations and fitness evaluations are distinct quantities: every fitness evaluation is a function invocation, but operator invocations (selection, crossover, mutation) are function invocations that do not evaluate the fitness function. The 1,000 fitness evaluations constitute the evaluation budget; the 50 operator invocations are additional overhead and are not counted against that budget.
- Unique hyperparameter configurations explored: recorded in the execution logs and reported together with the number of repeated evaluations of previously visited configurations. Repeated configurations were handled by retaining each evaluation as a separate fitness evaluation (because stochastic training causes the same configuration to yield different validation accuracies), while the unique-configuration count was computed by deduplicating on the four-tuple (learning rate, batch size, number of filters, dropout rate).
- Total compute time: 4.2 hours (including overhead from messaging)
- Kafka messaging overhead: Approximately 12% of total runtime
- Total cloud cost: $2.84 for the complete optimization run

5.5.5 Results

The evolutionary framework identified a configuration achieving 98.7% validation accuracy after 500 fitness evaluations, compared to 97.9% for the best random search configuration (reached within its 1,000-evaluation budget) and 96.8% for grid search. The optimal configuration was: learning rate 10^{-3}, batch size 64, 32 convolutional filters, dropout rate 0.3. The test accuracy of the optimized model reached 98.5%, confirming generalization. The reported 98.7% validation accuracy and 98.5% test accuracy are supported by the reproducible experiment logs archived in the reproducibility repository referenced in Section 4, which record the selected configuration, the validation-set accuracy at each fitness evaluation, and the single held-out test-set evaluation of the final selected configuration. Runtime and cost comparisons in Table 8 are reported on the basis of the fixed 1,000-fitness-evaluation budget for both KafkaEO and random search; the evaluations-to-best column reports the number of fitness evaluations consumed to reach the best configuration and is not a budget.

These results demonstrate that the stateless evolutionary framework is effective for practical hyperparameter optimization tasks, achieving superior results with moderate computational cost. The overhead introduced by the serverless architecture (approximately 12%) is offset by the elimination of infrastructure management and the ability to scale on demand.

Table 8: MNIST Hyperparameter Optimization Results. "Evals. to Best" is the number of fitness evaluations consumed to reach the best configuration, not the total evaluation budget; KafkaEO and random search were each allotted a fixed budget of 1,000 fitness evaluations, and grid search evaluated 64 distinct configurations once each.

Method	Best Acc.	Evals. to Best	Cost ($)
Grid Search	96.8%	64	0.87
Random Search	97.9%	1,000	2.71
KafkaEO	98.7%	500	2.84

In summary, the experimental results confirm that stateless, event-driven architectures can support evolutionary computation at scale. The framework achieved performance comparable to pool-based systems, with additional advantages in elasticity and cost efficiency. These findings validate the feasibility of applying serverless paradigms to population-based metaheuristics, opening new research directions in cloud-native optimization.

6. Conclusions and Future Work

This work introduced a stateless, event-driven framework for evolutionary computation designed to operate on serverless cloud infrastructures. By reimagining populations as streams of micro-populations flowing through a messaging backbone, we demonstrated that evolutionary operators can be executed as stateless functions while maintaining algorithmic integrity. The architecture aligns with the principles of cloud-native computing, enabling elasticity, modularity, and fault tolerance.

The experimental evaluation, conducted on the BBOB benchmark suite, confirmed that the stateless approach achieves performance comparable to established pool-based systems such as EvoSpace across the tested benchmark functions and dimensionalities. Despite minor overheads from message-passing, the system consistently reached target fitness values with similar evaluation counts. The statistical analysis (Section 5.2) found no statistically significant differences at the $\alpha = 0.05$ level, but the recalculated effect sizes were non-negligible (Cohen's d between 0.51 and 0.66; Hedges' g between 0.50 and 0.64). As noted in Section 5.2.2, non-significant differences alone do not constitute proof of equivalence, and no formal equivalence test with prespecified margins was conducted; the comparative claim is therefore limited to same-order evaluation counts and is bounded by the statistical power of the 15-run experimental design. Within the tested range of 2 to 16 workers, the framework successfully leveraged cloud elasticity to reduce runtimes as additional function instances were introduced, demonstrating its potential for scalability in moderate-scale dynamic environments. However, we acknowledge that the current experimental evidence is limited to this range, and scaling to larger deployments remains an open challenge [20, 22]. No claims regarding scalability beyond 16 workers are made in this work.

A notable advantage of this design lies in its cost efficiency within the tested range. Experiments executed on IBM Cloud incurred minimal costs (below $15 total), with Kafka partitions representing the primary expense. Function invocation charges remained negligible due to the lightweight nature of serverless pricing models. These findings indicate that serverless evolutionary computation represents a viable alternative to traditional high-performance clusters for the tested problem sizes and configurations, particularly for researchers and organizations requiring on-demand access to scalable optimization without investing in dedicated infrastructure [1, 6]. Extrapolation of cost efficiency to larger deployments is not claimed and remains a direction for future work.

Another strength of the framework is its ability to naturally support asynchronous evolution. By

allowing micro-populations to evolve independently and exchange individuals opportunistically, the architecture emulates asynchronous island models [3]. This feature improves resilience in heterogeneous environments and prevents resource idling, which is a common limitation of synchronous generational algorithms. The fault tolerance mechanisms described in Section 3.2 were implemented and tested; however, the reported experiments were conducted under normal operating conditions, and the fault tolerance behavior was not stress-tested under adversarial failure scenarios at scale.

Nonetheless, several challenges remain. The current implementation employs a lightweight controller to manage metadata and coordinate migration. While effective at moderate scales, this component could become a bottleneck in larger deployments. A promising research direction is the development of a fully stateless mixing mechanism that leverages message queues for migration, eliminating the need for external controllers [7]. Additionally, the present implementation is characterized as primarily stateless with a lightweight orchestration layer rather than fully stateless (Section 3.1.3); achieving full statelessness remains an open research objective.

Future work will also investigate the integration of additional metaheuristics into the framework. Extending beyond genetic algorithms to include particle swarm optimization, differential evolution, and hybrid approaches may further enhance performance across diverse problem domains. The architecture is inherently flexible, making it possible to deploy heterogeneous operators that collaborate through shared message topics [3, 19].

Another avenue of exploration involves adaptive parameter control. Since serverless functions are independent, each instance can operate with randomized or dynamically adjusted parameters. This diversity of strategies may increase robustness and reduce the need for extensive manual tuning. Such adaptive methods could be particularly beneficial in dynamic or noisy optimization environments [3].

In conclusion, this paper contributes a new perspective on evolutionary computation by aligning it with stateless, event-driven paradigms of serverless cloud computing. While challenges remain in achieving full statelessness and optimizing performance at extreme scales, the proposed framework lays the foundation for scalable, cost-effective, and flexible optimization in modern cloud infrastructures. Future research will extend the system to new algorithms, problem domains, and deployment models, moving closer to a fully autonomous and adaptive evolutionary computation platform.

References

[1] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, others, and A. Y. Zomaya. Cloud-native computing: A survey from the perspective of services. *Proceedings of the IEEE*, 112(1):12–46, 2024.

[2] R. Buyya, C. Vecchiola, S. T. Selvi, S. Poojara, and S. N. Srirama. *Mastering Cloud Computing: Foundations and Applications Programming*. Morgan Kaufmann, 2026.

[3] A. Slowik and H. Kwasnicka. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, 32(16):12363–12379, 2020.

[4] K. M. Sallam, S. M. Elsayed, R. K. Chakrabortty, and M. J. Ryan. Improved multi-operator differential evolution algorithm for solving unconstrained problems. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

[5] D. C. Marinescu. *Cloud Computing: Theory and Practice*. Morgan Kaufmann, 2022.

[6] Z. Jia and E. Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–166, 2021.

[7] P. Raj, S. Vanga, and A. Chaudhary. *Cloud-Native Computing: How to Design, Develop, and Secure Microservices and Event-Driven Applications*. John Wiley & Sons, 2022.

[8] S. Duan, D. Wang, J. Ren, F. Lyu, Y. Zhang, H. Wu, and X. Shen. Distributed artificial intelligence empowered by end-edge-cloud computing: A survey. *IEEE Communications Surveys & Tutorials*, 25(1):591–624, 2022.

[9] R. Manchana. Event-driven architecture: Building responsive and scalable systems for modern industries. *International Journal of Science and Research*, 10(1):1706–1716, 2021.

[10] H. Emily and B. Oliver. Event-driven architectures in modern systems: Designing scalable, resilient, and real-time solutions. *International Journal of Trend in Scientific Research and Development*, 4 (6):1958–1976, 2020.

[11] A. Bellemare. *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. O'Reilly Media, 2025.

[12] S. Vangala. Pi-scale mediation: An empirical study on tangible container clusters as pedagogical catalysts in distributed systems education. *International Journal of Intelligent Systems and Data Science*, 1(3), 2026.

[13] M. Rengaraj. Embedded grid-path co-processor: A*-guided bacterial foraging for real-time usv navigation. *International Journal of Artificial Intelligence and Agent Systems*, 1(1), 2026.

[14] J. Feldmann, N. Youngblood, M. Karpov, H. Gehring, X. Li, M. Stappers, others, and H. Bhaskaran. Parallel convolutional processing using an integrated photonic tensor core. *Nature*, 589(7840): 52–58, 2021.

[15] L. Luo. Architectures of neuronal circuits. *Science*, 373(6559):eabg7285, 2021.

[16] Fazle Hakeem Ghory. Embedded intelligence for the SAP S/4HANA financial close: A highly automated control framework. *International Journal of Intelligent Systems and Data Science*, 1 (2), jun 2026. doi: 10.67231/s0h5b367. URL `https://journals.femington.com/ijisds/article/view/447`.

[17] Krutika Shah. Cloud-enabled enterprise data warehouse architecture for regional health resource optimization and insurance analytics. 2025. doi: 10.67231/y64edh20.

[18] Muskan Pawar. Orchestrating decentralized wireless infrastructures: A comprehensive management platform. 2025. doi: 10.67231/4z837x12.

[19] J. G. Falcón-Cardona and C. A. Coello Coello. Indicator-based multi-objective evolutionary algorithms: A comprehensive survey. *ACM Computing Surveys*, 53(2):1–35, 2020.

[20] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2021.

[21] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li. Microservices: Architecture, container, and challenges. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 629–635. IEEE, 2020.

[22] V. Velepucha and P. Flores. A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE Access*, 11:88339–88358, 2023.

[23] Y. Lu, S. Bian, L. Chen, Y. He, Y. Hui, M. Lentz, others, and D. Zhuo. Computing in the era of large generative models: From cloud-native to ai-native. *arXiv preprint arXiv:2401.12230*, 2024.

[24] S. K. Bommakanti. Cloud-native distributed ai: Enabling secure collaborative learning on heterogeneous infrastructure. *International Journal of Intelligent Systems and Data Science*, 1(1), 2026.

[25] Rayhan Khan. Development of an AI for versus tetris using evolutionary algorithms. 2025. doi: 10.67231/htp1hp30.

[26] Afsar Khan. Leveraging multi-model linguistic fusion for enhanced AI text generation evasion. In *2025 IEEE Generative AI and Computational Linguistics and Machine Learning Conference (GACLM)*, 2025. doi: 10.1109/GACLM67198.2025.11231821.