FaaSBoard: Efficient Graph Processing with a Disaggregated Architecture on Serverless Services

YUSHI LIU* and YIKANG RUAN*, Shanghai Jiao Tong University, China
LETIAN RUAN, Shanghai Jiao Tong University, China
ZIJUN LI, Nanyang Technological University, Singapore
SEN GAO, National University of Singapore, Singapore
WEIHAO CUI, Shanghai Jiao Tong University, China
SHIXUAN SUN†, Shanghai Jiao Tong University, China
QUAN CHEN, Shanghai Jiao Tong University, China
SHUO QUAN, China Telecom Cloud Computing Research Institute, China
JIE WU†, China Telecom Cloud Computing Research Institute, China
BINGSHENG HE, National University of Singapore, Singapore
MINYI GUO, Shanghai Jiao Tong University, China

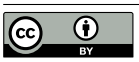
Graph processing workloads are increasingly being migrated to the cloud. With the growing adoption of serverless computing, graph processing gains advantages such as cost-effectiveness and resource elasticity. However, existing graph processing systems with monolithic function architecture struggle to detect intra-job resource elasticity and suffer from significant communication overhead.

In this paper, we present FaaSBoard, a graph processing system with a disaggregated serverless architecture powered entirely by serverless cloud services. FaaSBoard features a multi-tier data communication mechanism and an autonomous-elastic computing mechanism. Specifically, these two mechanisms are realized through image-based graph loading for faster starts, proxy-based collective communication leveraging high-bandwidth shared memory, 2D balanced graph partitioning for improved load balance, and a proactive terminate-and-respawn mechanism enabling fine-grained elasticity. Together, these four techniques collectively enhance both resource and overall execution efficiency. Experimental results demonstrate that FaaSBoard delivers up to 3.8× higher compute performance and reduces monetary cost by up to 61.5% compared to FaaSGraph, the current state-of-the-art serverless-based graph processing system. The source code of FaaSBoard is publicly available at https://github.com/SJTU-Liquid/FaaSBoard.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Mathematics of computing → Graph algorithms**.

*Both authors contributed equally to this research.
†Shixuan Sun and Jie Wu are the corresponding authors.

Authors' Contact Information: Yushi Liu, ziliuziliulys@gmail.com; Yikang Ruan, yiconrain@gmail.com, Shanghai Jiao Tong University, Shanghai, China; Letian Ruan, Shanghai Jiao Tong University, Shanghai, China, 1291903308rlt@sjtu.edu.cn; Zijun Li, Nanyang Technological University, Singapore, lzjzx1122@sjtu.edu.cn; Sen Gao, National University of Singapore, Singapore, sen@u.nus.edu; Weihao Cui, Shanghai Jiao Tong University, Shanghai, China, weihao@sjtu.edu.cn; Shixuan Sun, Shanghai Jiao Tong University, Shanghai, China, sunshixuan@sjtu.edu.cn; Quan Chen, Shanghai Jiao Tong University, Shanghai, China, chen-quan@cs.sjtu.edu.cn; Shuo Quan, China Telecom Cloud Computing Research Institute, Beijing, China, quansh@chinatelecom.cn; Jie Wu, China Telecom Cloud Computing Research Institute, Beijing, China, jiewu@temple.edu; Bingsheng He, National University of Singapore, Singapore, dcsheb@nus.edu.sg; Minyi Guo, Shanghai Jiao Tong University, Shanghai, China, guo-my@cs.sjtu.edu.cn.



ACM 2836-6573/2026/6-ART177
https://doi.org/10.1145/3802054

Additional Key Words and Phrases: Serverless Computing, Graph Processing, Disaggregated Serverless Architecture, Elasticity.

ACM Reference Format:
Yushi Liu, Yikang Ruan, Letian Ruan, Zijun Li, Sen Gao, Weihao Cui, Shixuan Sun, Quan Chen, Shuo Quan, Jie Wu, Bingsheng He, and Minyi Guo. 2026. FaaSBoard: Efficient Graph Processing with a Disaggregated Architecture on Serverless Services. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 177 (June 2026), 28 pages. https://doi.org/10.1145/3802054

1 Introduction

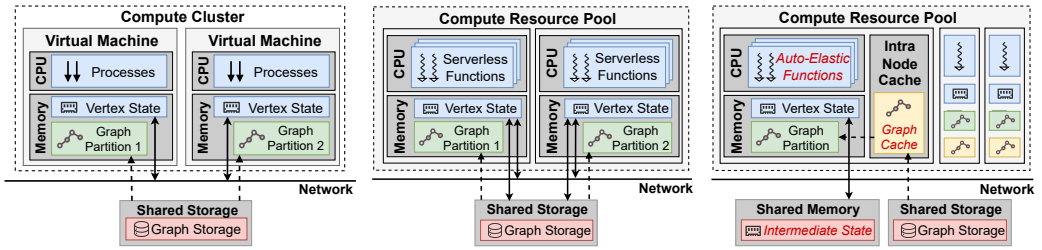
Graphs are powerful tools for modeling real-world entities and their relationships. Through everyday use of applications such as social networks, e-commerce platforms, and online banking, users generate numerous footprints that collectively form large-scale graphs, which companies can analyze to extract valuable business insights. As a result, graph processing has attracted significant attention, leading to the development of numerous frameworks [3, 4, 12, 17, 20, 37, 42, 58, 60, 66, 67] designed to accelerate computation by efficiently utilizing resources in and beyond a single server.

As graph datasets continue to grow in size, graph processing applications have increasingly shifted to the cloud [13, 40, 41] to conveniently leverage large-scale computing resources. Since most graph processing frameworks are originally designed for local clusters, a common approach is to replicate this setup in the cloud by provisioning cluster with monolithic servers. As illustrated in Figure 1a, the Graph Service Developer (i.e. the User of the Cloud) allocates a fixed number of VM instances, each with ample CPU and memory resources to host the graph processing framework.

Although the computation of a graph query follows a deterministic, iterative algorithm, real-world graph workloads exhibit highly dynamic job arrival patterns. Many graph workloads are issued as interactive, exploratory, or sporadic analytics queries [17, 46], where users submit jobs on demand and with little predictability in timing or concurrency [37, 54]. In such settings, traditional serverful graph processing systems suffer from limited resource elasticity: resources must be provisioned for peak demand and remain underutilized during idle periods, leading to inefficiency and increased cost.

Serverless computing offers a compelling alternative for handling this dynamism. By allowing resources to be provisioned on demand and released immediately after job completion, serverless platforms naturally support bursty and unpredictable workloads. Prior work has demonstrated that serverless functions can efficiently support large-scale data processing tasks while relieving users from explicit resource management [10, 18, 22, 31, 49, 53]. Moreover, the pay-as-you-go billing model aligns well with interactive analytics, where long idle periods between jobs are common. These benefits have motivated recent graph processing systems to adopt serverless architectures, using a monolithic function design in which a single function instance encapsulates the entire graph computation (Figure 1b). For example, FaaSGraph [37] integrates a FaaS runtime to orchestrate serverless functions for graph processing.

However, while such designs effectively address inter-query variability, our investigation shows that the current monolithic function architecture comes with trade-offs. First, the monolithic function architecture fails to exploit the inherent intra-query variability of graph processing. Graph workloads often exhibit fine-grained resource elasticity due to intra-iteration load imbalances, where faster compute instances must wait for others. Leveraging this intra-job elasticity remains challenging under a monolithic design due to the complex coordination required between functions. This leads to significant monetary waste from resource idling—averaging 36.4% in our profiling of FaaSGraph. Second, the communication of large data volumes becomes a critical bottleneck. The existing architecture is limited to either cloud storage or direct communication, suffering from



(a) Monolithic Server Architecture (b) Monolithic Function Architecture (c) Disaggregated Serverless Architecture

Fig. 1. Comparison of graph processing framework architectures in cloud environments. The dashed arrow represents graph loading, while the solid arrow indicates vertex state exchange. (a) Before graph processing begins, subgraphs are loaded into memory from shared storage. Vertex states are computed and exchanged over the network using broadcast and reduce. (b) A fleet of serverless functions replaces traditional servers for computation, with state exchanged via storage services (e.g., S3) or direct communication channels [14, 56]. (c) The graph processing system autonomously manages resource termination and respawning. Subgraphs are loaded via a cache layer, while vertex states are exchanged through a shared memory layer.

high communication overhead due to low bandwidth. These methods incur 1.43×-3.02× longer graph processing time compared to the optimal performance.

We identify the root cause as the architecture preserving the monolithic nature of resources, where functions are used like monolithic servers, but with limited resources and faster initialization. *We propose a disaggregated serverless architecture to address the deficiencies.* Figure 1c shows the architecture along with two critical mechanisms. First, a multi-tier data communication mechanism is introduced on top of a shared memory model enabled by the architecture. The shared memory provides a high-bandwidth communication channel to facilitate data transmission. Additionally, efficient data loading is supported by a tiered cache layer, where data is integrated closely with compute resource. Second, an autonomous-elastic computing mechanism is introduced, allowing resources to be terminated and spawned more frequently and proactively to exploit intra-job resource elasticity, while leveraging shared-memory to track query execution without losing progress.

Building on this design, we introduce FaaSBoard, a graph processing system that fully adopts a disaggregated serverless architecture. The user begins by partitioning the graph dataset, bundling with the execution binary into container images, and uploading to an image repository. Then, user can invoke queries seamlessly in the cloud without the need to provision or manage any servers. To this end, *FaaSBoard runs entirely in a serverless cloud ecosystem, eliminating the need for dedicated servers or long-running instances.* This design fully leverages the elasticity of serverless services, enhancing the system's generality and cost-efficiency.

In FaaSBoard, we propose four techniques across the data and compute layers to collaboratively enhance end-to-end system efficiency. Specifically, to implement the multi-tier data communication, it employs *Image-based Graph Loading*, which utilizes container images as a high-tier cache layer to enable efficient graph loading. Furthermore, *Proxy-based Collective Communication* leverages serverless containers as a shared memory layer to support high-bandwidth broadcast and reduce operations. On the compute side, FaaSBoard introduces the autonomous-elastic computing through a *Proactive Terminate-and-Respawn* mechanism, enabling autonomous resource management on serverless service. Furthermore, a *2D Balanced Graph Partitioning* method is introduced to improve graph processing performance under the resource constraints of serverless environments.

Our extensive experiments demonstrate that FaaSBoard achieves up to 3.8× higher graph processing performance and reduces monetary cost by up to 63.7%, compared to FaaSGraph [37], a state-of-the-art graph processing system adopting the monolithic function architecture. In summary, this paper makes three key contributions:

- We analyze the limitations of monolithic function architectures in graph processing, and motivate the design of a disaggregated serverless architecture.
- We introduce FaaSBoard, a graph processing system built on serverless cloud services to validate the effectiveness of the disaggregated architectural design.
- We conduct a comprehensive evaluation of FaaSBoard to demonstrate its superior performance and cost.

The remainder of the paper is organized as follows. Section 2 reviews background and related work. Section 3 motivates a disaggregated architectural design. Section 4 introduces the architecture and execution flow of the FaaSBoard system. Sections 5, 6 and 7 detail our implementation of such disaggregated architecture on serverless services. Section 8 presents a comprehensive evaluation to validate our design.

2 Background and Related Work

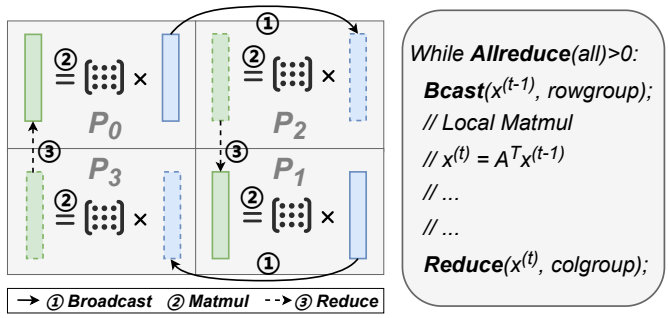
Serverless. Serverless computing has emerged as a compelling cloud service model, offering auto-scaling capabilities, a pay-per-use billing system, and a management-free experience for users. In this model, code packages are encapsulated as functions, and deployed by the cloud provider which handles request scheduling and resource scaling.

Many studies have adopted serverless computing as an approach to providing burstable computing resources, thereby facilitating large-scale data processing [10, 18, 31, 49, 53]. Due to the stateless nature of serverless functions, a common strategy for communication and data loading is to utilize shared storage services (e.g., S3, ElastiCache) [7, 35, 43–45]. However, this approach often results in performance degradation due to the high overhead and limited bandwidth of storage services. In FaaSBoard, we establish channels that are more tightly integrated with the serverless function service to optimize data transmission.

Some studies have proposed direct connection (e.g. Boxer [56], FMI [14]) to enable more efficient communication between functions. However, when only serverless functions are involved, communication performance remains limited due to the low bandwidth of individual functions. Other studies including Pocket [27] and Jiffy [26] utilize the EC2 cluster as a far memory pool to build a general-purpose storage system. In contrast, FaaSBoard is specifically designed to optimize collective operations common in graph processing by offloading reduce computations to a shared memory pool and leveraging high bandwidth to improve performance.

There are multiple ways to achieve fine-grained resource elasticity in serverless computing. For instance, small applications can be decomposed into microservices [21, 24, 32, 33] or workflows [34, 38, 39]. Data-intensive applications such as linear algebra [49] and SQL queries [23, 43, 45, 64] can be decoupled into small operators. However, our work focuses on supporting general graph processing algorithms, which are inherently difficult to decompose. Instead, we propose the autonomous elastic computing to leverage fine-grained elasticity through a resource-aware mechanism.

Graph Processing. A graph $G = (V, E)$ represents a set of vertices V and the edges $E \subseteq V \times V$ between them. Graph processing frameworks are designed to efficiently analyze large-scale graphs by iteratively applying user-defined algorithms to the graph and maintaining per-vertex state across iterations. For example, in single-source shortest path (SSSP), the framework computes the minimum distance from the source to every vertex. When a graph exceeds the capacity of a single worker, graph partitioning, which divides a graph into multiple subgraphs, becomes



(a) 2D Execution Pattern (b) Code Snippet

Fig. 2. Distributed Execution Model of FaaSBoard

essential to distribute computation across multiple workers while minimizing communication and maintaining load balance. Following prior work [37, 67], graph processing systems operate on partitions of the graph, with each worker processing a subgraph. Recently, graph processing has increasingly embraced serverless computing to exploit elastic resource provisioning. FaaSGraph leverages serverless elasticity to handle fluctuating workloads, mapping each graph partition to a serverless function acting as a worker. Unlike FaaSGraph [37], which modifies the serverless runtime, FaaSBoard targets existing serverless platforms without infrastructure changes, improving deployability and generality. Graphless [54] follows a similar infrastructure-agnostic approach, but its performance remains significantly behind that of modern graph processing systems.

3 Motivation

In this section, we motivate the need for developing a disaggregated architecture in serverless computing to streamline graph processing with enhanced efficiency and elasticity.

3.1 Linear Algebra based Execution Model

Following GraphBLAS [15], graph computations can be equivalently transformed into linear algebraic operations. More formally, iteration t of graph processing is represented as a sparse matrix-vector multiplication (SpMV) operation: $x^{(t)} = A^T x^{(t-1)}$, where A denotes the adjacency matrix of G, and $x^{(t)}$ represents the vector of vertex states at iteration t. The graph algorithm is determined by the multiplication $*$ and the addition + in matrix multiplication, which describe how to derive and aggregate vertex states. The programming model bears similarity to that of Graphite [42], allowing users to define functions that specify how to perform computations for neighboring vertex states.

Figure 2a illustrates a typical distributed execution pattern, where the adjacency matrix is divided into four submatrices, each assigned to a separate process. The blue vector from P_0, P_1 , representing the partitioned vertex states from the previous iteration, is broadcast within the row group to processes P_2 and P_3. Subsequently, each submatrix performs local computations to generate the green vector, which is then reduced within the column group to produce the result for the current iteration.

3.2 Deficiency of Monolithic Function Architecture

Figure 2b shows a code snippet that illustrates the execution model described in Section 3.1. We first implement this model on top of the existing monolithic function architecture and investigate

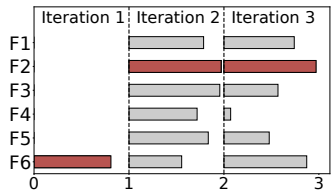


Fig. 3. Normalized execution time in three consecutive iterations of *tw*-BFS.

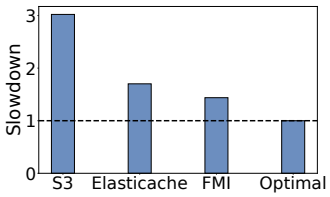


Fig. 4. Performance when applying various communication methods.

its behavior. All experiments are conducted using BFS on the Twitter [29] graph with around 1.8 billion edges. Constrained by limited memory of functions, the execution requires six functions.

Deficiency in Resource Elasticity. Due to the complex structure of graph data and irregular computation patterns, achieving perfect load balancing across workers is difficult. This leads to significant variation in execution times among functions in each iteration. Figure 3 shows the normalized execution time of six functions over three iterations, revealing substantial performance disparity. As BFS progresses and explores new frontiers, workloads shift dynamically—for instance, Function F2 is idle in iteration 1 but becomes heavily loaded in iteration 2. Such fluctuations complicate function behavior over time. In a complete execution, 36.4% of resources are wasted due to intra-iteration idle time, directly translating into equivalent monetary cost overhead.

However, serverless platforms cannot directly improve resource efficiency by utilizing the idle resources in each iteration because they lack visibility into application-level execution. A straightforward approach, terminating a worker after execution, falls short for graph processing, which requires iteration-based coordination among multiple workers. Additionally, workloads fluctuate across iterations, and functions may be invoked dynamically and unpredictably. Moreover, implementing such an approach often requires injecting custom logic into graph applications, leading to significant engineering overhead. Consequently, enabling fine-grained intra-job resource elasticity remains a major challenge for improving resource efficiency.

Observation 1. *In graph processing, worker execution times vary significantly across iterations, causing substantial resource underutilization. However, optimizing resource efficiency is challenging due to the need for coordinated, iterative execution across workers.*

Deficiency in Data Communication. Graph processing depends on collective operations to exchange vertex states, often involving large volumes of data. However, serverless functions have limited communication capabilities. We evaluate the impact of representative serverless communication mechanisms on BFS performance, including shared storage (e.g., S3, ElastiCache) and direct communication (e.g., FMI [14]). Figure 4 shows the slowdown compared to a baseline cluster with high-speed networking. The results reveal significant performance gaps, highlighting the inefficiency of bulk data transfers between serverless functions.[1]

Most existing systems use shared storage for communication [22, 35, 43–45, 49, 51], but performance suffers from low bandwidth and high latency [7, 14, 56]—a consequence of the decoupling between storage and compute. Additionally, since storage lacks compute capabilities, extra functions must be invoked to perform reduce operations [22], introducing further overhead. To address this, direct communication methods involving only serverless functions have been proposed [14, 56]. However, we find them suboptimal, as the limited bandwidth of serverless functions (75–90 MiB/s [7, 43]) necessitates binomial tree-style broadcast and reduce patterns [14], which

[1]Due to their poor performance, shared storage methods are excluded from later experiments in Section 8.3.2.

increase latency for leaf functions and add coordination overhead. Reduce operations also consume precious compute resources that could otherwise be used for graph processing. Moreover, we were unable to reproduce these methods on real-world platforms like AWS Lambda[2], suggesting they may be constrained by security, billing concerns, or susceptibility to infrastructure updates.

Observation 2. *Graph processing requires frequent bulk data exchange through collective operations, which serverless platforms handle poorly, resulting in performance degradation.*

3.3 Implication for System Design

Based on these observations, directly applying traditional cluster-based graph processing architectures to serverless platforms leads to limited resource elasticity and inefficient data communication. To address these challenges, we adopt a disaggregated architecture tailored for serverless environments. Specifically, we focus on two key mechanisms.

Autonomous-Elastic Computing. Current serverless graph processing systems scale only at the query level, limiting resource efficiency. Specifically, the complex coordination and the dynamic workload of functions hinder the system from exploiting per-iteration elasticity. To enable such fine-grained elasticity, we leverage a disaggregated architecture that provides shared memory for maintaining execution progress of the whole query independently of function lifecycle, allowing functions to terminate and respawn without losing progress. Building on this, we design an autonomously elastic computing mechanism that improves resource efficiency while minimizing decision-making overhead.

Multi-tier Data Communication. Communication in the monolithic function architecture is restricted to direct communication or shared storage, leading to overhead due to the absence of an intermediate approach. In contrast, our disaggregated system architecture utilizes a multi-tiered data hierarchy. Specifically, tiered caches can accelerate graph loading, while the shared memory pool enables efficient collective communication with high-bandwidth data transfer and built-in compute capabilities for reduce operations.

4 FaaSBoard Overview

We present FaaSBoard, a graph processing system that fully adopts the disaggregated serverless architecture in Section 3.3. Our core principle is the *exclusive reliance on serverless cloud services*, which guarantees scalability for every component, and makes our design orthogonal to the underlying cloud implementation, thereby enhancing the applicability. Figure 5 illustrates the architecture and execution flow of FaaSBoard.

Initialization. During the initialization phase, the user uploads a graph dataset along with the corresponding algorithm code. The graph is then partitioned into subgraphs using the *2D Balanced Graph Partitioner* (Section 6.1). Simultaneously, the code is compiled together with the helper graph runtime into an executable binary. The resulting subgraphs and the binary are packaged into images and submitted to a cloud image repository.

Execution. In the execution phase, after the user submits a query to the FaaS platform, a fleet of serverless functions is triggered, with each function processing a distinct subgraph. In cases of a hot start, graph data within each function is reused; otherwise, the system employs *Image-based Graph Loading* (Section 5.1) to load the graph partitions, using container image as a high-tier cache layer. During execution, the FaaS runtime implement the *Proactive Terminate-and-Respawn* mechanism (Section 6.2) to conserve resources during idle periods. Additionally, they utilize *Proxy-based Collective Communication* (Section 5.2) to perform collective operations such as broadcast and reduce, with proxies built atop the serverless container service acting as a shared memory layer.

[2]Our experiments simulate FMI using a proxy; see Section 8.3.2 for details.

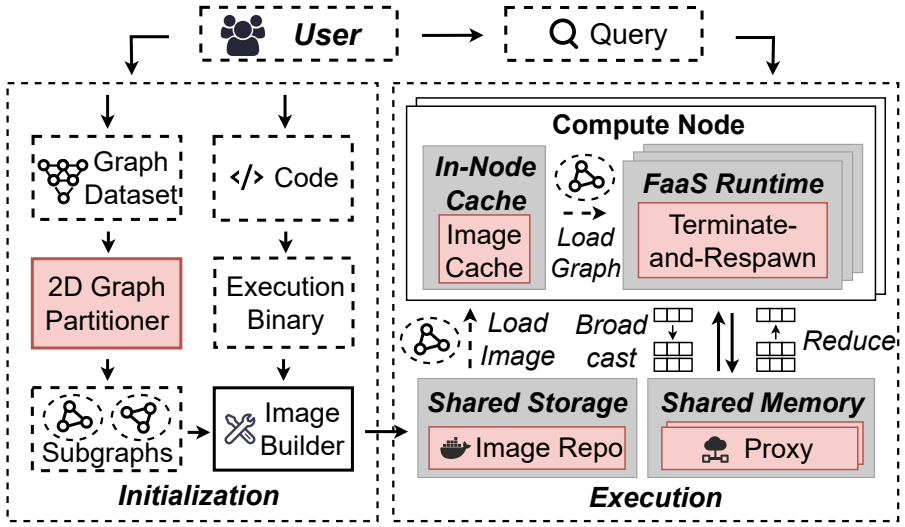


Fig. 5. FaaSBoard Architecture and Execution Flow. The gray components indicate adherence to the disaggregated serverless architecture in Figure 1c, while the red components highlight our proposed mechanisms.

Generality. In FaaSBoard, we focus on leveraging universal characteristics across major public cloud platforms. For instance, in Section 5.1, we introduce an image-based graph loading approach, exploiting the fact that major cloud providers like AWS [8] and Azure [1] often build image cache to accelerate image loading. Section 5.2 details our proxy-based communication, which utilizes the superior network bandwidth of container instances (e.g., AWS Fargate, Azure ACI, and GCP Cloud Run) over standard serverless functions. We choose AWS as our implementation platform because it is the industry benchmark and the most widely adopted public cloud, with the most mature serverless ecosystem. Although our design exploits common characteristics of serverless platforms, the current implementation is built and evaluated exclusively on AWS. Porting the system to other cloud platforms would require engineering effort to adapt platform-specific APIs and re-optimize communication and aggregation mechanisms.

In the following section, we will provide a detailed explanation of the mechanisms.

5 Graph Loading and Communication

In this section, we present the multi-tiered data communication supported by cloud services, incorporating efficient mechanisms for graph loading and vertex state exchange.

5.1 Image-based Graph Loading

Graph loading typically occurs during a cold start, when no computing resources are available and containers must be initialized from scratch. Users often rely on external storage services such as object storage (e.g., S3) or in-memory caching services (e.g., ElastiCache) for data storage. However, these approaches frequently suffer from performance inefficiencies mainly due to high network overhead and bandwidth limitations. As a result, we consider them to be the low-tier storage due to their complete separation from computing resources (i.e., serverless function service).

We propose that *loading data from container images is faster than from remote storage*, and argue that container images can serve as an ideal high-tier cache layer. In serverless function service, functions are packaged and executed within containers launched from these images. The images

encapsulate both the function code and all required software dependencies, which are loaded during a cold start. Cloud providers have been actively optimizing this loading process; for instance, AWS Lambda [8] employs multi-level caching to accelerate image loading, presenting a promising opportunity for high bandwidth data loading.

Thus, we propose our *Image-based Graph Loading* mechanism to benefit from such infrastructure. Specifically, after the graph is partitioned, each subgraph is bundled together with the execution binary into a container image. These images are then uploaded to the image repository service. During a cold start, rather than retrieving the subgraph from remote storage, the system directly loads the data from the local file stored within the container image. Although deploying container images may take tens of minutes, this cost is incurred offline and only once per graph version. In practice, this overhead is expected to be amortized across many function invocations, as the same graph snapshot is often repeatedly queried across multiple runs, including parameter sweeps [52] and concurrent multi-user workloads [59, 65]. As a result, image-based graph loading effectively shifts data access from remote storage to local image layers, significantly reducing cold-start data transfer overhead without impacting steady-state performance.

Our mechanism substantially reduces network overhead, while mitigating bandwidth bottlenecks by leveraging the local cache bandwidth across multiple compute nodes when functions are distributed. However, a challenge in fully utilizing this mechanism is the opacity of cache behavior to users, which makes graph loading performance unpredictable. To maintain the "hotness" of graph data, we introduce a *Data Ping* mechanism. This involves invoking functions at regular intervals (1 hour in FaaSBoard) to load graph data, ensuring persistent availability in the local cache, while incurring additional monetary costs due to repeated function invocations. Section 8.3.1 evaluates the loading performance and cost overhead in comparison to the shared storage approach.

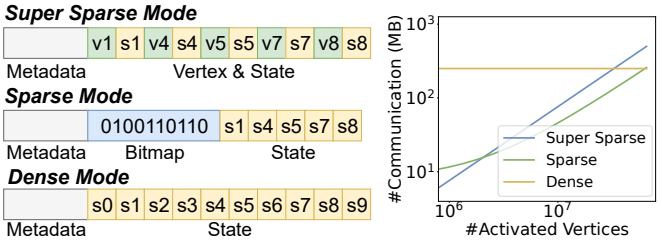
5.2 Proxy-based Collective Communication

Figure 2b shows the collective operations within FaaSBoard, where broadcast and reduce operations account for the majority of communication volume. In graph processing terms, the broadcast operation sends activated vertices and their associated states to other functions. Conversely, the reduce operation aggregates updated vertex states. Additionally, a vote operation (i.e. Allreduce) is employed to determine whether to compute in the next round. Next, we describe our *Proxy-based Collective Communication* mechanism, which focuses on efficient execution of collectives.

5.2.1 Proxy. The Proxy operates on top of the serverless container service (e.g., AWS Fargate). It acts as a shared memory layer, enabling fast data transfer between functions and overcoming the limitations of direct connection. Each function establishes a connection to a Proxy instance through a network socket at the beginning of a query, and disconnects once it is completed. Proxies can be multiplexed, allowing functions for different queries to connect to a single Proxy instance, thereby amortizing the monetary cost.

5.2.2 Collective Operations with Proxy. Next, we describe the implementation of collective operations using the Proxy. We employ *Vector* as the message abstraction.

Broadcast. During broadcast, the sender function first transmits the vector to the Proxy, which then distributes it to the receivers. Serverless containers offer substantially higher network bandwidth than functions. Specifically, using iPerf3 [2], we measured that each AWS Fargate container (16vCPU, 64GB memory) has a bandwidth of 636 MiB/s, which is 7.0X-8.5X greater than that of AWS Lambda. Thus, compared with direct connection approaches, vectors can be broadcasted in parallel to other functions, eliminating the need for a binomial tree routing and reducing network hops.



(a) Different Formats (b) #Communication

Fig. 6. Triple-mode Vector Format

Reduce. Result vectors from the senders are aggregated at the Proxy. Due to load imbalance, functions may generate their results at different times. Inspired by In-network Aggregation [30, 36, 47], an *In-memory Buffer* is introduced to support on-the-fly reduction, allowing vectors to be automatically aggregated upon arrival. Once the Proxy has collected all the results, it forwards the aggregated data to the receiver. Compared to direct connection, the Proxy offloads reduce computation from serverless functions, enabling concurrent execution of reduce and graph processing.
Vote. The Vote operation combines reduce and broadcast. The number of activated vertices for the next round is first reduced at the Proxy, then a decision is broadcast to all functions, indicating whether to continue computation.

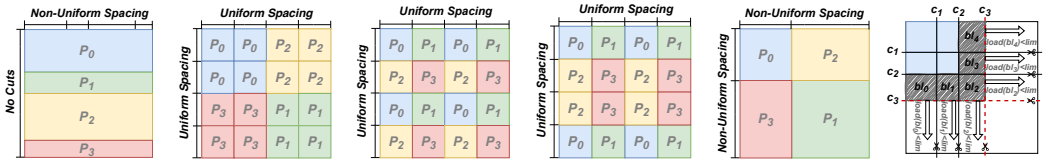
5.2.3 Triple-mode Vector Format. Figure 6a illustrates the different formats for Vector. For each format, metadata such as the message source and the number of companion vectors to wait for is prefixed. When the activated vertices are highly sparse, the vertices and their corresponding states are directly appended. If the activated vertices are moderately sparse, a bitmap is appended to indicate the activated vertices, followed by their states at the end. In scenarios where the activated vertices are dense, the states of all vertices are appended sequentially, regardless of their activation status.

Figure 6b illustrates the communication volume associated with various vector formats under different amount of activated vertices, using the Friendster graph [61] as an example. It demonstrates that adaptively choosing between formats can reduce communication overhead. For instance, selecting the Super Sparse Mode when the number of activated vertices is low proves beneficial. On the other hand, the Dense Mode is particularly advantageous as it is the only format that stores states in a contiguous order, enabling AVX Instruction Set-based optimizations at the Proxy server during reduce computations to improve performance. Additionally, Dense Mode is the only approach that enables zero-copy transmission, as FaaSBoard directly sends contiguous vertex states under this mode. In contrast, the other two formats require message reconstruction, introducing additional overhead due to the copying of vertex states.

We set the default thresholds for Super Sparse-Sparse and Sparse-Dense to 10^5 and 10^6, respectively. However, these thresholds also impact communication traffic under concurrent workloads, which we leave for future exploration. Section 8.3.2 highlights our performance advantages.

6 Graph Processing and Partitioning

In this section, we present our compute-oriented mechanisms, including a novel partitioning method that enhances graph processing performance under the resource constraints of serverless functions, and a Terminate-and-Respawn mechanism that enables autonomous-elastic computing.



(a) 1D [37, 67] (b) 2D-Grid [68] (c) 2D-Cyclic [4] (d) 2D-Staggered [3, 42] (e) 2D-Balanced (Ours) (f) *limit* Check

Fig. 7. Different chunk-based partition methods when $p = 4$, and the *limit* check example in our partition method.

6.1 2D Balanced Graph Partitioning

Design Rationale. Communication of large data volumes is a major performance bottleneck in serverless-based analytic systems [14, 34, 37, 63]. Accordingly, our system prioritizes communication efficiency, with graph partitioning playing a central role in reducing communication overhead while maintaining load balance.

From the matrix-based view of graph computation, partitioning strategies are commonly classified as *1D* or *2D* partitioning [19]. In *1D partitioning*, vertices (together with their incident edges) are assigned to workers along a single dimension by splitting rows or columns of the adjacency matrix (e.g., Figure 7a). Within this category, methods can be further divided into *chunk-based* and *min-cut–aware* approaches [25, 50]. Chunk-based methods partition vertices into contiguous ranges, offering low preprocessing overhead, while min-cut–aware methods explicitly minimize edge cuts at the cost of substantial computation and memory overhead.

Systems such as Gemini [67] and FaaSGraph [37] adopt 1D chunk-based partitioning rather than min-cut–aware strategies, as real-world graphs often exhibit strong structural locality [67]. Under such conditions, chunk-based partitioning preserves locality effectively while remaining significantly more lightweight. Compared to min-cut–aware approaches such as METIS [25], chunk-based methods drastically reduce preprocessing time and memory consumption during graph partitioning.

In contrast, *2D partitioning* divides the adjacency matrix into a grid of blocks (Figures 7b, 7c, and 7d), assigning each block to a worker. While 1D chunk-based methods follow an edge-cut model and incur communication complexity of $O(p)$ for broadcast and reduction operations, 2D partitioning reduces this cost to $O(\sqrt{p})$, where p is the number of partitions. As a result, many state-of-the-art graph processing systems [3, 4, 42] employ 2D chunk-based partitioning to improve communication efficiency.

However, existing 2D chunk-based approaches [3, 4, 42] rely on uniformly spaced cuts, which fail to account for graph sparsity and often lead to severe load imbalance. This issue is problematic in serverless environments, where each function is constrained by limited memory, and imbalance can easily trigger OOM failures. To address this limitation, FaaSBoard introduces a *2D balanced graph partitioning* strategy that simultaneously achieves computation locality, communication efficiency, and load balance, making it well suited for large-scale graph processing on serverless platforms.

Problem Formulation. We assume the graph G with n vertices and m edges will be partitioned into p subgraphs. We assume that p is a perfect square number, and relax this assumption in the subsequent discussion. We define the cut positions as $c_{0..\sqrt{p}}$ (where $c_0 = 0$, $c_{\sqrt{p}} = n$), making both vertical and horizontal cuts at the same positions to ensure that the diagonal submatrices

remain square. Our objective is to maximize load balancing, which is equivalent to minimizing the workload of the most heavily loaded submatrix. Equation 1 defines the workload of a submatrix, where A is the adjacency matrix, and nnz counts non-zero elements of the submatrix. The first and second elements represent computation and communication load, respectively. A coefficient β is employed to modulate the significance between computation and communication, where we assign $\beta = 10$ in FaaSBoard. Equation 2 formulates the optimization target.

$$load(A_{c_i:c_{i+1},c_j:c_{j+1}}) = nnz + \beta(c_{i+1} - c_i + c_{j+1} - c_j) \tag{1}$$

$$\min_{c} \max_{0<=i,j<\sqrt{p}} load(A_{c_i:c_{i+1},c_j:c_{j+1}}) \tag{2}$$

Step 1. Balanced Partitioning. Here, we propose a binary-search-based algorithm to generate the *optimal* partitioning under Equation 2. Specifically, we iteratively search for the minimum workload threshold, denoted as $limit$, such that there exists a partitioning of the graph where the workload in each subgraph does not exceed this threshold. If this condition is satisfied, we decrease the $limit$ and repeat the check; otherwise, we increase the $limit$. This process continues until the optimal $limit$ is determined, which at the same time generates the partition result (i.e. the cut positions c).

A key challenge is determining whether the $limit$ can be satisfied in polynomial time. To address this, we sequentially determine each cut position. Figure 7f illustrates an example. Suppose we have already determined positions c_1 and c_2. We initially set $c_3 = c_2$ and then incrementally move c_3 forward until one of the newly created submatrices $bl_{0..4}$ reaches the $limit$ at $c_3 = c'$. At this point, we fix c_3 to be $c' - 1$ and proceed to search for the next cut positions. If more than $\sqrt{p} - 1$ cuts are required to ensure that the workload of each subgraph remains below the $limit$, this indicates that the $limit$ cannot be satisfied. Otherwise, the $limit$ is feasible.

Algorithm 1 provides the pseudocode for our partitioning algorithm. The inputs include the adjacency matrix A, the required number of cuts $cut = \sqrt{p}$, and the error rate ϵ. Using a binary search approach, we iteratively refine the decision boundary $[left, right]$ and evaluate the workload $limit$ until either the optimal solution is found or the error rate falls below ϵ. For each $limit$, we generate cut positions $cuts$ and check if the number of cuts exceeds cut. For each vertex i, we process its incoming and outgoing edges, assigning them to the corresponding block cur_block. If the total edges in cur_block surpass $limit$, a new cut is fixed, cur_block is cleared, and the process continues until the final $cuts$ are determined.

The correctness is guaranteed by *Optimality*, ensuring it finds a valid partition result if one exists, and *Monotonicity*, which validates the effectiveness of the binary search.

Theorem 1. *Optimality: the algorithm produces minimal cuts for a given workload limit.*

Proof. Suppose there exists another partition plan $\hat{c}$ such that $|\hat{c}| < |c|$. Let k be the first position where $\hat{c_k} \neq c_k$. Firstly, $\hat{c_k} > c_k$ is impossible because the algorithm places c_k at the rightmost valid position without exceeding the $limit$. Secondly, if $\hat{c_k} < c_k$, moving the cut c_k leftward would leave a larger remaining subgraph, requiring at least $|c| - k$ cuts after c_k to satisfy the $limit$. This implies $|\hat{c}| >= |c|$, contradicting $|\hat{c}| < |c|$. Thus, we prove that c is optimal. □

Theorem 2. *Monotonicity: a valid partition plan for limit implies one for limit′ > limit; no valid plan for limit implies none for limit′ < limit.*

Proof. We omit the proof as it is obvious. □

The time complexity of this algorithm is $O(mlog(m + n))$, which accounts for the binary search and the $limit$ check. Although this is higher than the complexities of previous methods ($O(n)$ for

Algorithm 1: Balanced Partitioning Algorithm

Input: A adjacent matrix in CSR format, cut number of cuts, ϵ error rate.
Output: $cuts$ generated cuts.

```
left, right, cuts := 1, A.edges, [];
while (right − left)/right > ϵ do
    limit, valid, cur_cuts := (left + right)/2, true, [0];
    block, cur_block := [], [];
    for i=0..N − 1 do
        cur_block.clear();
        for src in i.inedges where src ≤ i do
            id := cur_cuts.size × 2 − 2 − between_id(cur_cuts, src);
            cur_block[id] + +;
        end
        for dest in i.outedges where dest ≤ i do
            id := between_id(cur_cuts, dest);
            cur_block[id] + +;
        end
        for j=0..cur_cuts.size × 2 − 2 do
            if block[j] + cur_block[j] > limit then cur_cuts.push(i); block := cur_block;
            else block += cur_block;
        end
        if cur_cuts.size > cut then valid := false; break;
    end
    if valid then cuts, left := cur_cuts, limit + 1;
    else right := limit − 1;
end
```

1D-based and $O(1)$ for 2D-based), we consider it feasible since the partitioning algorithm only needs to run once. Furthermore, we introduce two optimizations to improve its performance. First, we implement an early-stop mechanism in the binary search instead of pursuing the optimal result. In our system, we tolerate a 1% error for the $limit$, as this has minimal impact on graph processing performance while significantly reducing the number of binary search iterations. Second, we parallelize the binary search process. Specifically, assuming we have t threads, we divide the current decision boundary for $limit$ into $t + 1$ equally spaced intervals and evaluate the t possible $limit$ values in parallel. This approach allows us to quickly narrow down the decision boundary and find the result efficiently.

Step 2. Binpacking. In FaaSBoard, Users should be able to adjust the number of subgraphs p with flexibility, which influences the number of functions and CPU resources. To enable tuning of the latency-cost trade-off, we propose a binpacking algorithm to support flexible adjustments of non-square p. First, the graph is partitioned into $\lceil\sqrt{p}\rceil^2$ balanced partitions. Then, partitions are iteratively merged in pairs to maximize communication savings, reducing the count to p. For instance, submatrices with interleaved rows or columns are merged. The process repeats, prioritizing pairs with the highest communication gain, until p subgraphs remain.

Merging subgraphs reduces communication overhead but increases the risk of load imbalance. In particular, the workload of a merged subgraph may exceed that of the largest subgraph, violating the optimization objective in Equation 2. To mitigate this, we introduce an additional input parameter provided by user, $ratio$, which constrains merging such that the combined workload remains within

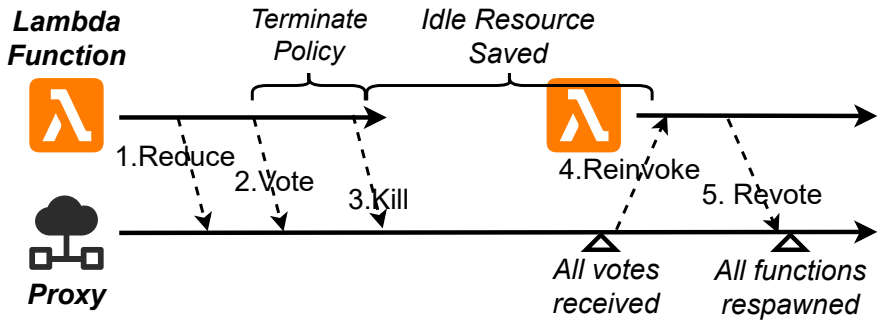


Fig. 8. Terminate-and-Respawn Mechanism.

ratio times the average workload across all subgraphs. In FaaSBoard, the system is configured by default to generate the most possible balanced subgraphs. This is achieved by iteratively fine-tuning the *ratio* until it reaches its minimum possible value. The binpacking algorithm operates efficiently with minimal overhead, as it processes no more than a few dozen graph partitions given our datasets.

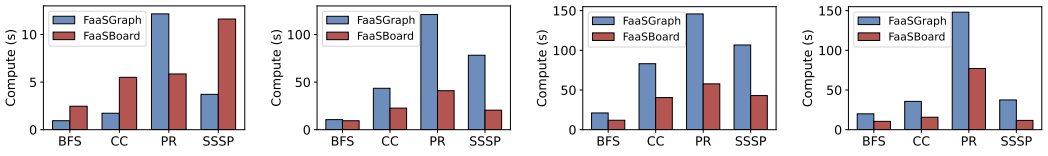
The effectiveness and efficiency of our algorithm are analyzed in detail in Section 8.3.3.

6.2 Proactive Terminate-and-Respawn

Although the algorithm in Section 6.1 aims to optimize load balancing, achieving perfect load balance is often impractical. The challenge arises primarily due to the power-law degree distribution commonly found in real-world graph datasets [20], where a small subset of vertices possesses extremely high degrees. As a result, resources may remain idle, creating opportunities for optimization. Serverless, along with FaaSBoard's execution model, enables us to leverage this opportunity. As shown in Figure 2a, functions off the diagonal receive the vector at the start and send responses at the end without retaining vertex state across iterations. Serverless is ideal for such stateless computations, where functions can be spawned and terminated in each iteration without disrupting the overall query execution. This allows functions to utilize computational resources only for the necessary duration, thereby reducing unnecessary costs.

Here, we introduce our *Proactive Terminate-and-Respawn* mechanism. The execution flow of this mechanism is shown in Figure 8. Specifically, after sending the reduce vector, functions send a vote message to the Proxy and wait for others to complete. Rather than idling until the next iteration begins, they autonomously decide to exit the function execution based on a *Terminate Policy*. This termination is communicated via a kill message, which the Proxy uses to track the number of terminated functions. After some time, when the Proxy determines it has received sufficient vote messages, it reinvokes all terminated serverless functions. Each respawned function sends a revote message to the Proxy to signal its restoration. Once all terminated functions have been respawned, the next iteration proceeds as usual.

A key challenge in this mechanism is designing an effective Terminate Policy, in which the decision-making time is significantly smaller than the resource idle time for more cost savings. Due to the dynamic workload, the policy should also be able to detect appropriate load imbalance opportunities for termination, and avoid unnecessary terminations that could introduce latency overhead when the workload is balanced. Here, we propose a lightweight Terminate Policy. The idea is that if a function remains idle for a prolonged period, it is likely to stay idle for even longer. To implement this, we assign a grace period of T milliseconds to each function after its sending of



(a) Livejournal (b) Twitter (c) Friendster (d) Rmat27

Fig. 9. Compute Time Comparison.

the vote message. If all votes are received within this T ms window, the function continues to run. However, if the grace period elapses (i.e. the function doesn't receive the response of vote in T ms), the function is terminated. In FaaSBoard, we set the default T to 300 ms.

Users don't need to modify the execution process shown in Figure 2b, as the implementation is internally handled within the vote operation. Specifically, if the Allreduce returns a timeout signal, the function is terminated immediately. Upon reinvocation, the function retries the Allreduce to send a Revote message and resumes execution seamlessly, as if no termination had occurred.

While latency overhead may occur due to cold starts during function reinvocation, where resource initialization and subgraph reloading take place, this is often mitigated in practice. For most graph applications, the termination period between iterations is typically less than one minute. This short duration increases the likelihood that the released function resources remain alive in the cluster, thanks to the keep-alive policies commonly employed by major cloud providers [48]. As a result, the container and subgraph data can often be reused without reinitialization.

An alternative approach is to spawn functions in proportion to the number of active vertices at each iteration. However, this requires functions to dynamically fetch the corresponding graph data at runtime. Because active vertices are typically scattered across multiple partitions, this approach triggers frequent and redundant data transfers, substantially increasing latency in serverless environments. In contrast, our approach is partition-aware: functions are respawned directly on the partitions that host the active vertices for the next iteration. As a result, each function reloads only the minimal necessary state, preserving data locality across iterations and avoiding redundant data movement.

Section 8.3.4 presents the analysis of the cost savings and latency overhead linked to this mechanism.

7 Implementation

Tools. We developed the FaaSBoard system using approximately 9.6K LOC of C++ code. The Proxy is built using epoll, enabling it to continuously manage connections, send and receive messages, and aggregate intermediate vertex states. We implement the dual-mode message passing as well as the work stealing mechanism from Gemini [67] in FaaSBoard. Furthermore, we leverage the AVX2 instruction set for optimized processing of our uint32 data, operating on 256-bit batches.

Parallel Strategy. As shown in Figure 2a, a function must perform the broadcast-compute-reduce sequence for each subgraph. To enhance execution efficiency, we employ distinct parallel strategies for each operation, carefully tailored to their respective computational characteristics and resource demands. Specifically, we use graph parallelism for broadcast and reduce, enabling all communications to occur simultaneously to fully leverage network bandwidth. In contrast, we apply vertex parallelism for compute, allowing concurrent edge scanning for vertices but processing each subgraph sequentially to prevent compute resource contention.

Table 1. Hardware, software setup and benchmarks.

	Configuration			
Hardware	*FaaSGraph:* CPU: Intel Xeon E5-2676 V3 @2.40Ghz, Instance: m4.10xlarge, Cores: 40, DRAM: 160GB			
Software	*FaaSGraph:* Ubuntu 22.04, Golang 1.18.4, Docker 20.10.17 *FaaSBoard:* Image public.ecr.aws/lambda/provided:al2023			
Service	*FaaSBoard:* AWS Lambda, AWS Fargate			
Function	*FaaSGraph&FaaSBoard:* Cores: 2, DRAM: 3538MB			
Benchmark	Graph	Vertices	Edges	#Functions
	Livejournal	4.84M	68.9M	1(BFS,CC,PR,SSSP)
	Twitter	41.6M	1.46B	6(BFS,PR),12(CC,SSSP)
	Friendster	65.6M	1.80B	8(BFS,PR),14(CC,SSSP)
	Rmat27	134M	2.14B	9(BFS,PR),17(CC,SSSP)

Proxy Scaling. Unlike Kubernetes, which supports scaling based on customized resource metrics, AWS Fargate only allows scaling based on CPU or memory utilization. This creates a mismatch with the network-intensive nature of the Proxy. Therefore, FaaSBoard adopts the Bounded Scaling Policy from FaaSGraph [37] for managing Proxies. Each Proxy is assigned a defined *capacity*, representing the maximum number of concurrent queries it can handle to avoid network contention. This capacity is computed by dividing the available bandwidth by the maximum potential message size (i.e., assuming the Dense format). For each incoming query, the system decides whether to queue it at an existing Proxy or to scale up a new one, aiming to minimize the estimated waiting time. Benefiting the capacity-aware the proxy instances are typically warm and available before functions scale out, and their startup overhead is amortized over many concurrent invocations. Unused Proxy is automatically reclaimed after remaining idle for 15 minutes.

Fargate does not violate the disaggregated design of FaaSBoard. The proxy is not involved in graph computation but only serves as a shared, disaggregated communication and coordination layer that enables efficient data exchange among independently scaled serverless functions. Graph computation is still fully executed by stateless functions, and compute, storage, and communication remain logically separated. Therefore, the proxy does not reintroduce centralized graph processing but rather provides lightweight, elastic remote memory for inter-function communication.

Fault Tolerance. Fault tolerance is critical for long-running jobs (e.g., day-scale executions). In contrast, the graph workloads are often short-lived, e.g., minutes-scale jobs. For such short executions, a simple retry-based recovery mechanism is typically sufficient and incurs very low overhead in practice, avoiding the substantial system complexity required by full-fledged fault-tolerant protocols. Moreover, serverless platforms inherently discourage long-running tasks, where statically provisioned clusters are a more appropriate choice.

8 Evaluation

This section validates the disaggregated serverless architecture by evaluating the performance of FaaSBoard.

8.1 Setup

Baseline Systems. We use FaaSGraph [37], a state-of-the-art serverless graph processing system as our baseline for overall evaluation. To compare partitioning strategies, FaaSBoard implements many graph partitioning methods from prior researches [3, 4, 37, 42, 67]. Additionally, we integrate FMI [14] to evaluate communication approaches. We do not include classic distributed graph

systems (e.g., Gemini) in our comparison because our work focuses on elasticity in serverless environments, while these systems assume fixed clusters. Their performance has already been evaluated in prior serverless graph studies such as FaaSGraph.

Hardware and Software. Table 1 summarizes the experimental configurations. FaaSBoard is deployed fully on AWS Lambda and AWS Fargate. Since FaaSGraph relies on a custom serverless runtime, we deploy it on an EC2 instance. To select the instance type, we evaluate CPU performance across AWS instance generations (e.g., m7, m6, m5, m4), and discover that the m4 series offers performance closest to Lambda, leading us to choose m4.10xlarge for our experiments. To simulate an environment identical to AWS Lambda, we disable FaaSGraph's shared-CPU and shared-memory. However, all containers run on the same instance to measure the upper-bound performance.

Resource Config. FaaSBoard adopts the same configuration as FaaSGraph, allocating 2 vCPUs and 3 GB of memory per function. For each graph benchmark, we use FaaSGraph's Resource Optimizer to determine the number of functions, with details listed in Table 1[3]. We deploy the Proxy on serverless container service, where each container is configured with 16 vCPUs and 64 GB of memory.

Pricing Strategy. We adopt AWS's pricing model. In FaaSBoard, the total monetary cost comprises two components: (i) the cost of all AWS Lambda functions, which are employed as our workers; and (ii) the cost of AWS Fargate, which we introduce as proxy to handle communication.

Let M_l denote the cost of running all AWS Lambda functions and M_f the cost of running AWS Fargate. The cost M_l is calculated as $M_l = t \times m_l \times c_l$, where m_l is the total memory (in GB) allocated to all Lambda functions, and c_l is the per-second-per-GB cost for Lambda. The AWS Fargate cost M_f, following its dual-component pricing model, is given by $M_f = t \times (v_f \times c_v + m_f \times c_m)$, where v_f and m_f are the total number of vCPUs and the memory (in GB) allocated for Fargate, respectively; c_v and c_m are the per-second-per-vCPU and per-second-per-GB pricing coefficients for Fargate, respectively. Here, t stands for the end-to-end execution time of a graph computing task. The total monetary cost for FaaSBoard is thus given by:

$$M = M_l + M_f = t \times m_l \times c_l + t \times (v_f \times c_v + m_f \times c_m) \tag{3}$$

For the x86 architecture, c_l is set to \$0.0000166667 per second. The Fargate pricing coefficients c_v and c_m are set to \$0.000014 (\$0.05056 per hour) and \$0.000001536 (\$0.05056 per hour), respectively, based on the price list in AWS.

We maintain AWS Lambda's pricing for FaaSGraph, assuming it is a serverless service and follows the pay-as-you-go billing. However, the billing for FaaSGraph corresponds solely to the M_l component within the FaaSBoard cost model, with no additional charges incurred for the proxy.

Graph Benchmarks. We use four widely adopted graph datasets from prior work [3, 37, 42, 67]: LiveJournal(*lj*) [6], Twitter(*tw*) [29], Friendster(fr) [61] and RMAT-27(rm) [11]. The graph sizes are summarized in Table 1. We select four standard graph workloads: Breadth-First-Search (BFS), PageRank (PR), Connected-Components (CC) and Single-Source-Shortest-Path (SSSP)[4].

8.2 Overall Comparison

Time. Figure 9 presents the in-memory compute time of FaaSGraph and FaaSBoard. The results show that FaaSBoard performs differently on small and large graphs. For *tw*, *fr* and *rm*, FaaSBoard outperforms FaaSGraph by 1.1×-3.8×. This highlights that, despite the overhead in serverless cloud services, FaaSBoard can still perform better in performance. With FaaSBoard, we benefit both from

[3]CC runs on an undirected graph, while SSSP runs on a weighted graph. Their CSR size is larger than the directed graph used by BFS and PR and thus needs extra functions to process.

[4]BFS and SSSP use vertex 0 as the source. For PR, we execute 20 iterations.

Table 2. Monetary cost (¢) comparison between FaaSGraph and FaaSBoard.

Dataset	System	BFS	CC	PR	SSSP
lj	FaaSGraph	**0.01**	**0.01**	**0.07**	**0.02**
	FaaSBoard	0.09	0.21	0.22	0.44
tw	FaaSGraph	**0.37**	3.01	4.18	5.40
	FaaSBoard	0.63	**2.30**	**2.74**	**2.08**
fr	FaaSGraph	0.97	6.70	6.72	8.60
	FaaSBoard	**0.93**	**4.10**	**4.53**	**4.36**
rm	FaaSGraph	1.04	3.49	7.67	3.67
	FaaSBoard	**0.88**	**2.04**	**5.49**	**1.52**

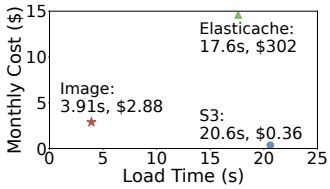


Fig. 10. Load time and monthly monetary cost.

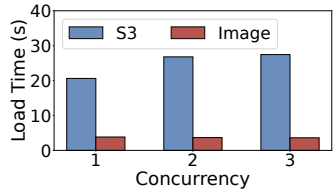


Fig. 11. Load time under different concurrency.

the improved performance brought by the disaggregated serverless architecture and the operational simplicity provided by cloud services.

However, FaaSGraph outperforms FaaSBoard for small graphs like *lj*. This is because FaaSGraph follows a subgraph-centric model, performing all local updates to convergence within each subgraph before global synchronization. In contrast, FaaSBoard adopts an SpMV-style model, synchronizing after each update round. As a result, FaaSGraph requires fewer iterations (e.g., *lj*-BFS: 5 vs. 17). As a result, interactions with the Proxy or Coordinator are minimized, leading to lower synchronization overhead. This is further confirmed by the performance of PageRank, where FaaSBoard remains faster than FaaSGraph when both run the same number of iterations. Therefore, FaaSBoard is suitable for processing large graphs.

Cost. Table 2 reports the monetary cost computed using Equation 3. On *tw*, *fr*, and *rm*, FaaSBoard achieves lower cost in nearly all benchmarks, reducing monetary cost by 4.1%–61.5%. An exception is *tw*-BFS, where FaaSBoard incurs a 70.3% higher cost due to proxy overhead combined with slower execution compared to FaaSGraph.

On the small *lj* graph, FaaSGraph is more cost-efficient, as the workload fits within a single function. In this case, the fixed Fargate overhead increases the total cost without providing communication benefits, and FaaSGraph also executes faster than FaaSBoard. Overall, these results are consistent with our efficiency analysis and show that, for one-shot queries on large graphs and compute-intensive workloads, FaaSBoard provides superior cost efficiency.

8.3 Detailed Evaluation

In this subsection, we examine each design choice and provide a detailed analysis on performance and cost efficiency. Unless specified otherwise, we follow all resource and execution configurations outlined in Section 8.1.

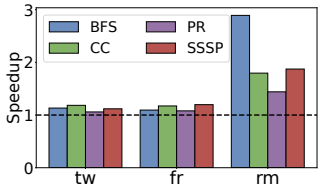


Fig. 12. Speedup achieved by applying Proxy over FMI.

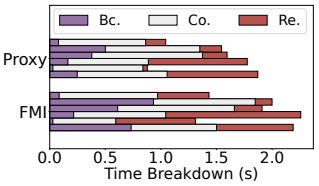


Fig. 13. Execution breakdown. (Bc: Broadcast, Co: Compute, Re: Reduce)

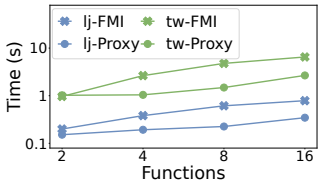


Fig. 14. Performance of sending vector to different number of functions.

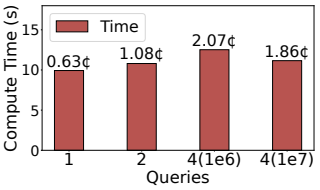


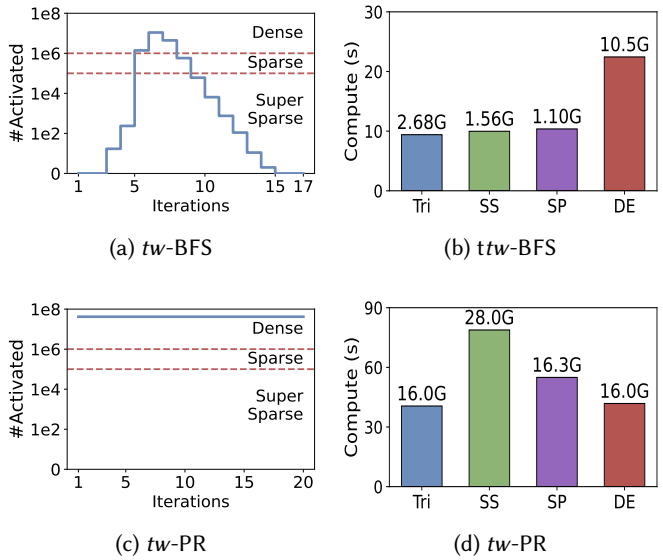
Fig. 15. Concurrent *tw*-BFS with a single Proxy. The number indicates cost.

8.3.1 Graph Loading. We begin by analyzing our graph loading mechanism. In our evaluation, we measure the loading time of *fr*-BFS, where 8 functions run concurrently to load the subgraphs. Figure 10 presents the loading time and estimated monthly cost for different storage approaches. The loading time is averaged over 10 consecutive runs, with a 1-hour interval between each run for the image-based approach. S3 and ElastiCache involve only storage costs, while the image-based approach addtionally involves the cost of invoking functions hourly over a month. We observe that the image-based approach significantly improves loading performance with only a slight increase in cost compared with S3. Additionally, the image-based approach delivers stable load performance with a CV of 0.088. ElastiCache is the most expensive option due to its memory-based design.

Figure 11 illustrates the load time for 1, 2, and 3 concurrent *fr*-BFS jobs. The load time is measured from the start of the invocation to the completion of loading of the last function. As concurrency increases, we observe that the load time for S3 increases due to its bandwidth bottleneck, while the image-based approach maintains stable load performance, as it can leverage bandwidth across multiple servers.

8.3.2 Collective Communication. In this subsection, we assess the Proxy-based communication by comparing it with FMI [14]. As we cannot reproduce FMI's direct function connection behavior in AWS Lambda, we simulate it in FaaSBoard by implementing function-to-function communication via Proxy for send and receive operations. On top of this, we implement FMI's binomial tree broadcast and reduce.

Figure 12 illustrates the compute time speedup on *tw*, *fr*, and *rm* when applying Proxy compared to FMI, achieving a speedup of 1.05×-2.89×. Proxy-based communication has a significant impact on larger graphs like *rm*, as they are split into more partitions, resulting in a deeper binomial tree in FMI. Furthermore, the increased number of vertices amplifies communication volume. Figure 13 presents the execution breakdown of a single PR iteration on the *tw* graph. The communication accounts for a substantial portion in FMI setting, whereas it is significantly reduced in Proxy setting.



(a) *tw*-BFS (b) t*tw*-BFS

(c) *tw*-PR (d) *tw*-PR

Fig. 16. Performance analysis of the Trimode vector format. Numbers indicate total communication volume (SS: Super Sparse, SP: Sparse, DE: Dense).

Next, we conduct a microbenchmark to examine the impact of shallow communication structure when using Proxy. We broadcast vectors of varying lengths to different numbers of functions and measure the broadcast time. The vector length is computed as $\#vertices * 4$ bytes for graphs *lj* and *tw*, which differ by approximately an order of magnitude. As shown in Figure 14, Proxy consistently outperforms FMI when using more than four functions. As the number of functions increases, Proxy maintains relatively stable performance due to its high bandwidth, whereas FMI exhibits a increase in broadcast time due to its deeper tree structure. However, at 16 functions, Proxy's network bandwidth becomes saturated, indicating the need for further scaling.

Next, we use *tw*-BFS to demonstrate the multiplexing capability of Proxy and the benefits of the trimode vector format. Figure 15 presents the compute time and monetary cost for running concurrent queries with a single Proxy, measured from invocation to the completion of the last query. We observe that the monetary cost increases sublinearly with the number of queries, as the cost of Proxy is amortized. However, at four concurrent queries, we experience a cost rise due to suboptimal performance. This is caused by the high communication volume generated by the dense format, leading to network congestion. To validate this, we increase the Sparse-Dense threshold to 10^7, rerun the experiment, and observe a reduction in both latency and cost.

Trimode Format. We evaluate the effectiveness of the Trimode vector format. As shown in Figure 16a and 16c, the demand for Super Sparse, Sparse, and Dense vector formats varies significantly across different applications. For example, in *tw*-BFS, across 17 iterations, the algorithm uses the Dense format in 3 iterations, the Sparse format in 1 iteration, and the Super Sparse format in the remaining iterations, whereas *tw*-PR uses the Dense format across all iterations. Figures 16b and 16d compare the execution time of *tw*-BFS and *tw*-PR using Trimode versus using a single vector format exclusively. In *tw-BFS*, the number of activated vertices fluctuates dynamically; as most iterations exhibit sparsity, using Super Sparse or Sparse formats results in lower communication volume and reduced execution time. In contrast, *tw*-PR activates all vertices in every iteration, making the

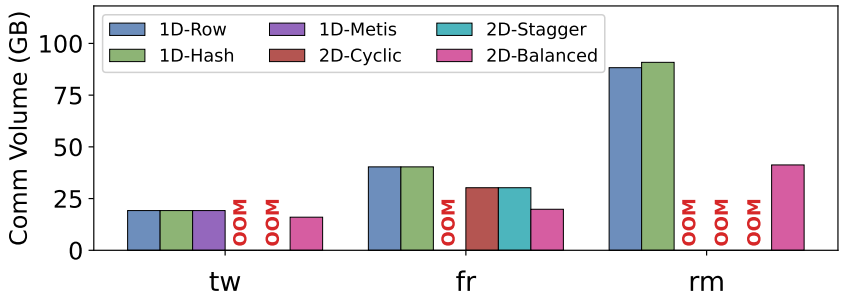


Fig. 17. Communication volume of PR under different partitioning strategies.

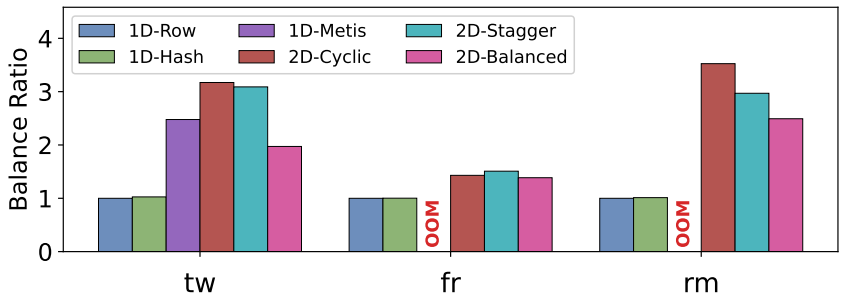


Fig. 18. Balance ratio of different partitioning strategies.

Dense format superior in terms of both communication efficiency and computational performance. Overall, these results demonstrate that a mixed-mode vector representation is essential to achieving optimal performance across diverse graph applications and varying iteration characteristics.

8.3.3 Graph Partitioning. We evaluate the impact of different graph partitioning methods, including *1D partitioning* schemes (Row [37, 67], Hash, and METIS [25] [5]) and *2D partitioning* schemes (Cyclic [4], Stagger [3, 42], and the Balanced method proposed in this paper). All partitions are generated offline on a machine equipped with 512 GB memory and 128 CPU cores. METIS requires 2660 seconds to partition the *tw* graph and fails with OOM errors on the larger *fr* and *rm* graphs. In contrast, all other methods complete efficiently within practical time (a few minutes) and memory budgets. We execute FaaSBoard using partitions produced by each method. Table 3 reports PR execution time, Figure 17 shows the communication volume, and Figure 18 reports the load balance ratio, defined as the maximum number of edges assigned to any partition divided by the average number of edges per partition.

As discussed in Section 6.1, 2D partitioning methods can significantly reduce communication volume compared to simple 1D schemes such as Row and Hash. However, this reduction often comes at the cost of load imbalance. While Row and Hash achieve excellent balance ratios by evenly distributing edges across partitions, they incur substantially higher communication overhead.

[5]We integrate the hash-based scheme (assigning vertex x to partition x%p) and METIS-based scheme into FaaSBoard by re-ordering vertices according to the partition results before chunking them.

Table 3. Comparison of PageRank (PR) compute time (s) under different partitioning strategies.

Graph	*1D*			*2D*		
	Row	Hash	METIS	Cyclic	Stagger	Balanced
tw	45.6	**36.1**	37.9	OOM	OOM	43.1
fr	66.0	60.3	OOM	60.3	63.3	**57.7**
rm	224.0	116.5	OOM	OOM	OOM	**77.1**

Table 4. Performance comparison of 2D-Grid versus 2D-Balanced partitioning for PageRank (PR).

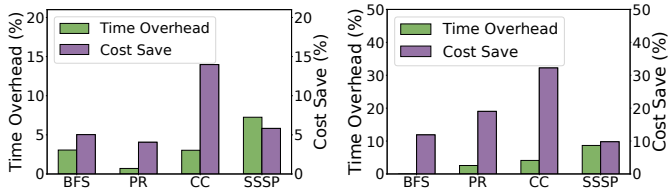
Graph	*2D-Grid*			*2D-Balanced*		
	Time	Comm	Balance	Time	Comm	Balance
tw	45.5	**19.2**	4.78	**34.1**	**19.2**	**2.97**
fr	85.6	30.2	**1.38**	**57.1**	**21.7**	2.86
rm	OOM	OOM	7.22	**77.1**	**41.2**	**2.49**

Existing 2D chunk-based methods, including Cyclic and Stagger, reduce communication but fail to account for the uneven sparsity of real-world graphs. As a result, some partitions may contain excessive numbers of edges, causing individual serverless functions to exceed memory limits and trigger OOM failures. This limitation makes these methods impractical in serverless environments. Our balanced 2D partitioning method explicitly considers edge distribution during partitioning. By jointly optimizing communication volume and partition balance, it avoids oversized partitions while retaining the communication efficiency of 2D layouts. As shown in our evaluation, this design achieves a favorable balance ratio without sacrificing communication efficiency, leading to consistently strong end-to-end performance.

Min-cut–based partitioning performs well on *tw*, as it explicitly optimizes both balance and edge cuts. However, its high computational and memory overhead severely limits scalability: METIS requires hours of preprocessing and fails to scale to larger graphs such as *fr* and *rm*.

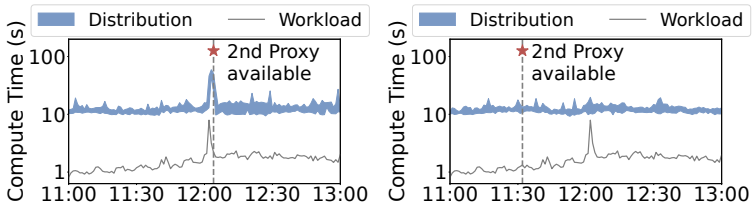
We also considered the partitioning scheme used in GridGraph [68] as shown in 7b. However, GridGraph primarily targets out-of-core computation, aiming to reduce I/O operations, which requires the number of partitions to be a perfect square. In contrast, FaaSBoard is designed as a distributed system. Following GridGraph's default configuration, we scale the number of functions for *tw*-PR and *fr*-PR to 9 (the nearest larger perfect square) for our experiments; the results are summarized in Table 4. Since Grid partitioning shares the uniform spacing technique utilized by Cyclic and Staggered methods (Figures 7b, 7c, and 7d), it is susceptible to severe load imbalances. This imbalance leads to OOM failures on the *rm*-PR and higher communication volume on *fr*. Consequently, our 2D-Balanced method consistently outperforms the 2D-Grid scheme in terms of compute time.

8.3.4 Terminate-and-Respawn. In this subsection, we evaluate the compute time overhead and cost savings of our mechanism compared to disabling it, using *tw* as the example graph dataset. Figure 19a shows that with default graph partitioning and the most balanced subgraphs, our mechanism achieves 4.0%-13.9% cost savings at the cost of 0.7%-7.2% execution overhead. This indicates that idle resources still exist with our load-balanced partitioning, and the Terminate-and-Respawn mechanism serves as a backup to further optimize resource and reduce costs. In Figure 19b, where we set the $ratio$ from Section 6.1 to 2, cost savings increase significantly because of more unbalanced workload, while the compute overhead is negligible in comparison.



(a) Balanced Partitioning (b) Unbalanced ($ratio = 2$)

Fig. 19. Compute time overhead and cost saving under Terminate-and-Respawn mechanism running on *tw*.



(a) Proxy on Fargate (b) Proxy with Faster Initialization

Fig. 20. Compute time distribution under real-world workload. The number of Proxy is scaled from 1 to 2, with the exact time of scaling marked by a red star.

8.4 Trace Analysis

In this section, we evaluate our performance using a 2-hour trace from FaaSGraph [37] on peak hour. We select *tw*-BFS as our example graph workload. Since the graph size for *tw* is approximately 100X larger than the dataset analyzed in FaaSGraph, we scale down the original trace by a factor of 100 to ensure the experiment fits within our budget.

Figure 20a shows the compute time distribution over 2 hours. We observe a surge in compute time around 12:00, caused by the extended time to scale up a Proxy in Fargate, which takes approximately 43 seconds. Before query spike arrives, following the scaling policy in FaaSGraph [37], queries tend to queue to avoid the high proxy scaling overhead. However, when too many queries accumulate, a Proxy is finally scaled, resulting in even further delays as the FaaSBoard system response slowly in handling the surging workload.

Compared to Fargate's container scaling time, we measure the time required to spawn a Proxy container on our local server to be approximately 1 second. To simulate a lower container initialization overhead in Fargate, we conduct an experiment where all proxies are kept alive but become available to the query scheduler after 1 second. The performance is shown in Figure 20b. We observe that FaaSBoard maintains steady performance even under fluctuating workloads. Notably, the second Proxy is scaled up more proactively due to its low overhead, and it is quickly deployed to handle graph processing queries. This highlights the importance of optimizing the initialization time of serverless container services on the cloud, in addition to function services, which can be addressed via many approaches such as checkpointing [5, 16, 28, 55] and container caching [9, 33, 62].

8.5 Performance under Different Graph Orderings

Real-world graphs typically follow a *power-law degree distribution* and exhibit significant sparsity. The datasets used in our evaluation—including *lj*, *tw*, *fr*, and *rm*—all exhibit these characteristics. To assess the robustness of FaaSBoard under different graph layouts, we evaluate three vertex ordering strategies: (1) Random Ordering, which randomly permutes vertex IDs and eliminates structural locality; (2) Natural Ordering, which follows the vertex IDs provided by the original datasets and the graph ordering used in our experiments; and (3) Gorder [57], which reassigns vertex IDs to place frequently co-accessed vertices close together, improving spatial locality.

Figure 21 shows that FaaSBoard exhibits consistent performance trends across different graph applications: Gorder achieves the best performance, followed by natural ordering, while random ordering performs the worst. Importantly, FaaSBoard maintains strong performance under natural ordering for most graph applications, demonstrating robustness to graph layout variations.

Although reordering techniques such as Gorder can improve performance by enhancing spatial locality, the performance gap between natural ordering and Gorder is small for several applications. This is because real-world graphs often already exhibit substantial inherent locality. Moreover, Gorder introduces non-trivial preprocessing overhead; for example, reordering the *tw* graph from natural ordering to Gorder takes approximately 1.5 hours. In practice, this creates a trade-off between offline reordering cost and the achievable runtime performance benefits.

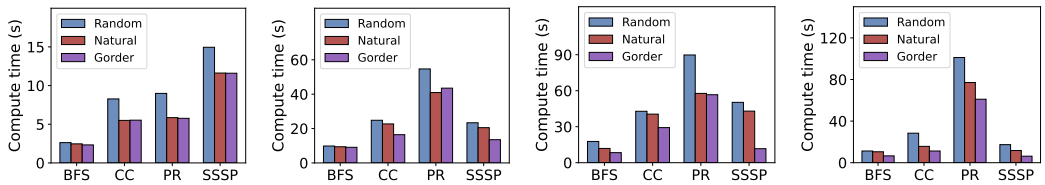


Fig. 21. Compute time on different Graph-Apps under different graph orderings.

9 Conclusion and Future Work

We present FaaSBoard, a graph processing system on serverless cloud services which adopts a disaggregated architectural design, integrating an autonomous-elastic computing mechanism with a multi-tier data system to optimize both resource and execution efficiency. Experimental results show that FaaSBoard significantly outperforms FaaSGraph in terms of both compute performance and monetary cost. Currently, our approach targets static or slowly evolving graphs, such as daily updates or periodically generated graph snapshots. When the graph is updated, a new container image is rebuilt and deployed offline; this process can be executed asynchronously and overlapped with ongoing analytics on the previous version, enabling near-instant switching at runtime. However, workloads with very frequent graph updates (e.g., second-level changes in highly dynamic graphs) are not yet supported, as repeated image construction would become a bottleneck. Extending our approach to efficiently support such highly dynamic graphs is an interesting direction for future work.

Acknowledgments

The work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62572303.

References

[1] [n. d.]. Azure artifact cache. https://learn.microsoft.com/en-us/azure/container-registry/artifact-cache-overview.

[2] [n. d.]. iPerf. https://iperf.fr.

[3] Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud, Mucahid Kutlu, Mostafa Shehata, and Tamer Elsayed. 2018. LA3: A scalable link-and locality-aware linear algebra-based graph analytics system. *Proceedings of the VLDB Endowment* 11, 8 (2018), 920–933.

[4] Michael J Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L Willke, and Pradeep Dubey. 2016. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 313–322.

[5] Lixiang Ao, George Porter, and Geoffrey M Voelker. 2022. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 730–746.

[6] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*. ACM, 44–54. doi:10.1145/1150402.1150412

[7] Thomas Bodner, Theo Radig, David Justen, Daniel Ritter, and Tilmann Rabl. 2025. An Empirical Evaluation of Serverless Cloud Infrastructure for Large-Scale Data Processing. *arXiv preprint arXiv:2501.07771* (2025).

[8] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 315–328.

[9] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[10] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. 13–24.

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[12] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[13] Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. 2012. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*. 1–13.

[14] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. Fmi: Fast and cheap message passing for serverless functions. In *Proceedings of the 37th International Conference on Supercomputing*. 373–385.

[15] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.

[16] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[17] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[18] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 363–376.

[19] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. 2018. A study of partitioning policies for graph analytics on large-scale distributed platforms. *Proceedings of the VLDB Endowment* 12, 4 (2018), 321–334.

[20] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[21] Zhipeng Jia and Emmett Witchel. 2021. 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*. 152–166.

[22] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*. 857–871.

[23] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Ditto: Efficient serverless analytics with elastic parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 406–419.

[24] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2023. Executing microservice applications on serverless, correctly. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 367–395.

[25] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

[26] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. 2022. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 697–713.

[27] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 427–444.

[28] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective checkpoint orchestration for serverless hot-starts. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 298–316.

[29] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*. ACM, 591–600. doi:10.1145/1772690.1772751

[30] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 741–761.

[31] Yilong Li, Seo Jin Park, and John Ousterhout. 2021. MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 593–611.

[32] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. 2020. Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 399–408.

[33] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing. In *2022 USENIX annual technical conference (USENIX ATC 22)*. 69–84.

[34] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 782–796.

[35] Mengfan Liu, Wei Wang, and Chuan Wu. 2025. Optimizing distributed deployment of mixture-of-experts model inference in serverless computing. *arXiv preprint arXiv:2501.05313* (2025).

[36] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray CC Cheung, and Jianfei He. 2023. In-network aggregation with transport transparency for distributed training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 376–391.

[37] Yushi Liu, Shixuan Sun, Zijun Li, Quan Chen, Sen Gao, Bingsheng He, Chao Li, and Minyi Guo. 2024. Faasgraph: Enabling scalable, efficient, and cost-effective graph processing with serverless computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 385–400.

[38] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. 2021. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 285–301.

[39] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 303–320.

[40] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. ACM, 135–146. doi:10.1145/1807167.1807184

[41] Jasmina Malicevic, Amitabha Roy, and Willy Zwaenepoel. 2014. Scale-up graph processing in the cloud: Challenges and solutions. In *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*. 1–6.

[42] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. 2020. Graphite: A NUMA-aware HPC system for graph analytics based on a new MPI* X parallelism model. *Proceedings of the VLDB Endowment* 13, 6 (2020), 783–797.

[43] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD*

Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020. ACM, 115–130. doi:10.1145/3318464.3389758

[44] Matthew Perron, Raul Castro Fernandez, David Dewitt, Michael Cafarella, and Samuel Madden. 2023. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.

[45] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 131–141. doi:10.1145/3318464.3380609

[46] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[47] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with {In-Network} aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 785–808.

[48] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 205–218.

[49] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless linear algebra. In *Proceedings of the 11th ACM symposium on cloud computing*. 281–295.

[50] George M Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. 2017. Partitioning Trillion-edge Graphs in Minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 653–669.

[51] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: a serverless vector data management system using cloud functions. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[52] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 15)*. 425–440.

[53] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 2021. Dorylus: Affordable, scalable, and accurate {GNN} training with distributed {CPU} servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 495–514.

[54] Lucian Toader, Alexandru Uta, Ahmed Musaafir, and Alexandru Iosup. 2019. Graphless: Toward serverless graph processing. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 66–73.

[55] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*. 559–572.

[56] Michael Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data analytics on network-enabled serverless platforms. In *11th annual conference on innovative data systems research (CIDR 2021)*. ETH Zurich.

[57] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.

[58] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices* 50, 8 (2015), 194–204.

[59] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. 2014. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC 14)*. 227–238.

[60] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[61] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*. IEEE Computer Society, 745–754. doi:10.1109/ICDM.2012.138

[62] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 335–350.

[63] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1489–1504.

[64] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus:{NIMBLE} task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 653–669.

[65] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 441–452.

[66] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[67] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.

[68] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.

Received October 2025; revised January 2026; accepted February 2026