electronics

MDPI

Article

Comparative Evaluation of Java Virtual Machine-Based Message Queue Services: A Study on Kafka, Artemis, Pulsar, and RocketMQ

Md Showkat Hossain Chy [1], Muhammad Ashfakur Rahman Arju [2], Sri Manjusha Tella [1] and Tomas Cerny [3,*]

[1] Computer Science, Baylor University, One Bear Place #97741, Waco, TX 76798, USA; mdshowkathossain_ch1@baylor.edu (M.S.H.C.); srimanjusha_tella1@baylor.edu (S.M.T.)
[2] Computer Science, Montana University, Culbertson Hall, 100, Bozeman, MT 59717, USA; muhammadashfaku.arju@student.montana.edu
[3] Systems and Industrial Engineering, University of Arizona, 1127 East James E Rogers Way #111, Tucson, AZ 85721, USA
* Correspondence: tcerny@arizona.edu

Abstract: Message Queue (MQ) services play a vital role in modern distributed systems as they enable asynchronous communication between services and facilitate the decoupling of various components of the system. Among the many MQ services available, Kafka, Apache Pulsar, Artemis, and RocketMQ are popular choices, each offering unique features and capabilities. As the adoption of MQ services continues to grow, choosing the appropriate service that can meet the requirements of the system has become increasingly challenging. Therefore, a comprehensive comparison of these services is crucial to determine the most suitable one for a specific use-case. This research paper presents a thorough evaluation of these MQ services based on critical metrics such as CPU utilization, memory usage, garbage collection, latency, and throughput. Based on our extensive review, no other research has delved into such a detailed evaluation, thereby establishing our work as a cornerstone in this field. The results of our study offer valuable insights into the strengths and limitations of each service. Our findings indicate that each message queue behaves differently inside the Java Virtual Machine (JVM). This work aims to assist developers and researchers in strategically deploying and optimizing MQ services based on specific system and use-case requirements. In addition to providing machine metrics, our results demonstrate the performance of each message queue under different load scenarios, making it a valuable resource for those seeking to ensure the effective functioning of their MQ services.

Keywords: message queue; Kafka; Pulsar; Artemis; RocketMQ; JVM; latency; throughput; CPU utilization; garbage collection

Citation: Chy, M.S.H.; Arju, M.A.R.; Tella, S.M.; Cerny, T. Comparative Evaluation of Java Virtual Machine-Based Message Queue Services: A Study on Kafka, Artemis, Pulsar, and RocketMQ. *Electronics* **2023**, *12*, 4792. https://doi.org/10.3390/electronics12234792

Academic Editor: Paulo Ferreira

Received: 18 October 2023
Revised: 22 November 2023
Accepted: 23 November 2023
Published: 27 November 2023

1. Introduction

As we transcend into an era dominated by complex digital ecosystems, the significance of asynchronous communication within modern distributed systems cannot be overstated. This intricate web of interactions necessitates a robust communication framework capable of bridging the divide between various system components while preserving core attributes such as scalability, reliability, and decoupling [1]. Message queue (MQ) services stand out as essential in this realm, fueling seamless communication between various services. Their pivotal role lies in facilitating asynchronous communication and decoupling within distributed systems. Among the myriad of MQ services available, Kafka [2], Pulsar [3], Artemis [4], and RocketMQ [5] have been identified as particularly significant due to their widespread adoption and distinct architectural differences, which render them ideal for a comprehensive analysis.

The choice of Kafka, Pulsar, Artemis, and RocketMQ as subjects of this study is not merely due to their popularity but also stems from their diverse architectural approaches

and the different challenges they address in distributed environments. Each of these MQ services has been designed with unique features that cater to specific requirements of modern-day applications, making them leaders in the field. For instance, Kafka is renowned for its high throughput and scalability, Pulsar for its multi-tenancy and geo-replication features, Artemis for its lightweight nature and performance, and RocketMQ for its reliability and broad integration capabilities. This diversity in capabilities and design philosophies provides a rich basis for comparison and analysis.

As the demand for MQ services continues to surge across industries, the task of selecting the most appropriate service for a specific use-case has become increasingly intricate [6]. The plethora of options available can be perplexing for developers and architects seeking to optimize their systems. Therefore, the necessity of conducting a comprehensive comparison of these MQ services has grown more apparent with each passing day [7]. This research paper embarks on the formidable journey of conducting an in-depth evaluation of Kafka, Pulsar, Artemis, and RocketMQ, with a primary focus on critical performance metrics, including CPU utilization, memory consumption, garbage collection efficiency, latency, and throughput. Our objective is to unravel the nuanced strengths and limitations of each service, thereby providing a valuable resource for developers, architects, and researchers navigating the complex terrain of distributed systems.

A noteworthy revelation stemming from our rigorous research is the profound divergence in behavior exhibited by each message queue service when operating within the confines of the Java Virtual Machine (JVM). This stark variation underscores the critical importance of bespoke selection and optimization to cater to specific use-cases and systemic prerequisites. By illuminating the idiosyncrasies of these MQ services, our study contributes to the strategic decision-making process associated with message queue service deployment and optimization.

The findings of our evaluation provide developers and system architects with a deeper understanding of the strengths and limitations of each MQ service, thereby enabling them to make informed decisions when selecting the appropriate service for their specific use-case. In addition to machine metrics, our evaluation also includes the performance of each message queue under different load scenarios, which can help to identify the appropriate service for a specific use-case, especially where the load may fluctuate over time. To the best of our knowledge, no other study has undertaken such a comprehensive analysis, thus making our research a benchmark in this domain.

In our comprehensive evaluation, we harnessed the versatile capabilities of the OpenMessaging Benchmark Framework [8]—a meticulously designed performance testing tool developed under the auspices of The Linux Foundation. This tool, meticulously crafted, stands as the linchpin of our endeavor for several compelling reasons.

First and foremost, the OpenMessaging Benchmark Framework was tailor-made to navigate the complex landscape of messaging systems. It has the innate ability to traverse the intricate nuances of messaging systems, regardless of the diversity of workloads and scenarios. Its adaptability is a remarkable asset, seamlessly accommodating a myriad of messaging protocols and message queues. This feature was pivotal, providing us with a unified and flexible testing framework that proved indispensable for our comprehensive assessment.

Our choice of the OpenMessaging Benchmark Framework goes beyond its capabilities. It embodies a commitment to consistency in our benchmarking process. By allowing us to evaluate all four message queues without bias or partiality towards specific technologies, it safeguards the integrity of our research findings. The tool's resonance within the developer community, as evidenced by its impressive statistics on GitHub and widespread acceptance OpenMessaging Benchmark Framework (https://github.com/openmessaging/benchmark, accessed on 18 October 2023). This popularity reaffirms our confidence in its efficacy as an essential instrument for our research. In summary, the OpenMessaging Benchmark Framework was a deliberate and strategic choice, serving as both a

powerful instrument for our evaluation and a testament to our commitment to rigorous, unbiased research.

The subsequent sections of this manuscript meticulously delineate our exploration into message queue benchmarking: Section 2 introduces the unique facets of each benchmarked message queue and their architectures; Section 3 situates our study within the context of related works; the benchmarking methodology is unfolded in Section 4; Sections 5 and 6 delineate the conducted experiments and their resultant data; discussions and insights are encapsulated in Section 7; Section 9 describes some settings or environments that can negatively affect our approach, i.e., threats to validity; and finally, Section 10 concisely summarizes our findings and conclusions.

2. Background

In this section, we summarize the key features and capabilities of the selected four message queue systems for this study: Kafka, Pulsar, Artemis, and RocketMQ. Table 1 provides a brief comparison of the key differences between these MQs. The main objective of this paper is to analyze the performance of each messaging queue service in terms of critical metrics such as CPU usage, memory, latency, and throughput, which provides us with insight into how data flows within these distributed architectures.

Table 1. Key features of message queue systems.

Message Queue	Key Features
Kafka	• Fault-tolerant, scalable, high-throughput solution for processing and transmitting real-time data. • Publish–subscribe model connecting producers and consumers. • Horizontal scalability for handling vast message volumes through distributed architecture.
Pulsar	• Unique architecture with separated serving and storage layers for high-throughput, low-latency messaging. • Tiered storage optimization based on access patterns and retention policies. • Fine-grained access control system enhancing security.
Artemis	• Robust open-source message queue supporting various messaging patterns. • Modular architecture which is customizable and facilitates seamless communication between producers and consumers. • Offers advanced features such as durable subscriptions, message grouping, and clustering capabilities for high availability.
RocketMQ	• Originally designed for Alibaba's e-commerce ecosystem and evolved into a versatile solution for high-throughput and fault-tolerant messaging. • Publish–subscribe model, supporting multiple messaging patterns. • Strength in handling real-time streaming data with architecture-supporting data pipelines.

2.1. Kafka

Apache Kafka [2] is one of the most popular and notable in the realm of distributed streaming platforms. It was significantly developed to address the challenges in handling vast volumes of real-time data. Kafka's design provides a fault-tolerant, scalable, high throughput solution for processing and transmitting the streams of records efficiently.

Kafka seamlessly connects the producers and consumers in a publish–subscribe model. It enables the producers to dispatch records to topics, serving as conduits for data streams. Kafka's architecture, as demonstrated in Figure 1 [6], ensures fault-tolerant and distributed storage of records, safeguarding data integrity in case of node failures. Another key feature is its horizontal scalability, which can effortlessly handle a vast number of messages per second through its distributed architecture that partitions data across multiple brokers, enabling parallel processing. Kafka's unique topic partitioning allows for concurrent processing and high scalability.

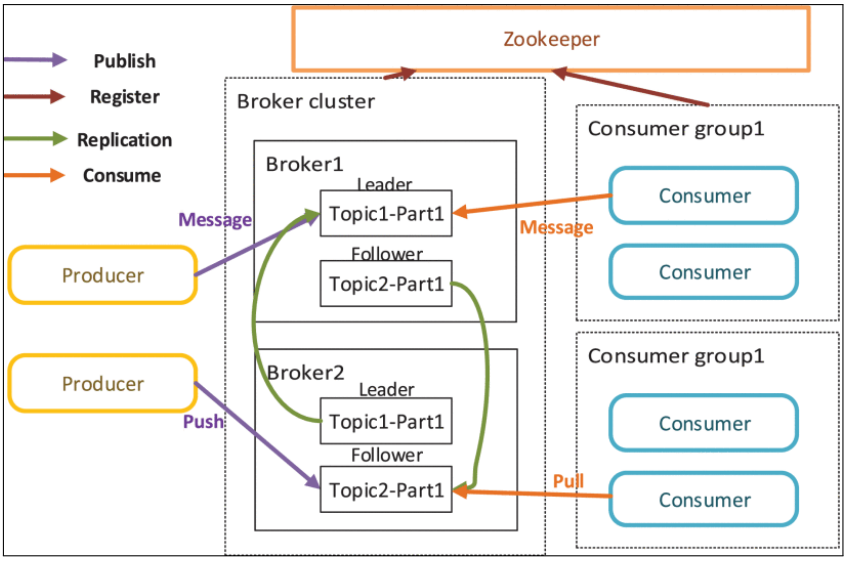


Figure 1. Kafka's architecture [6].

2.2. Pulsar

Apache Pulsar [3] was developed to tackle the challenges of high-throughput and low-latency messaging in the realm of distributed messaging and event streaming platforms. It offers a versatile solution for handling diverse data-intensive applications. Pulsar distinguishes itself through a unique architecture that separates serving and storage layers, providing an agile and scalable foundation for real-time data processing.

Apache Pulsar's features make it effective for data-intensive apps. Notably, Pulsar introduces a tiered storage architecture, optimizing data storage by organizing it across different tiers based on access patterns and retention policies. This innovative approach ensures efficient resource utilization. Pulsar further emphasizes security through a fine-grained access control system, allowing administrators to define precise access policies at various levels. Its proficiency in handling high-throughput workloads makes it a reliable choice for rapid and scalable data processing in demanding scenarios.

Pulsar's event stream processing capabilities set it apart, enabling the creation of dynamic and responsive event-driven architectures. The platform's multi-tenancy support ensures efficient resource sharing while maintaining isolation, a crucial feature for cloud environments. Pulsar's architecture is displayed in Figure 2 [6].

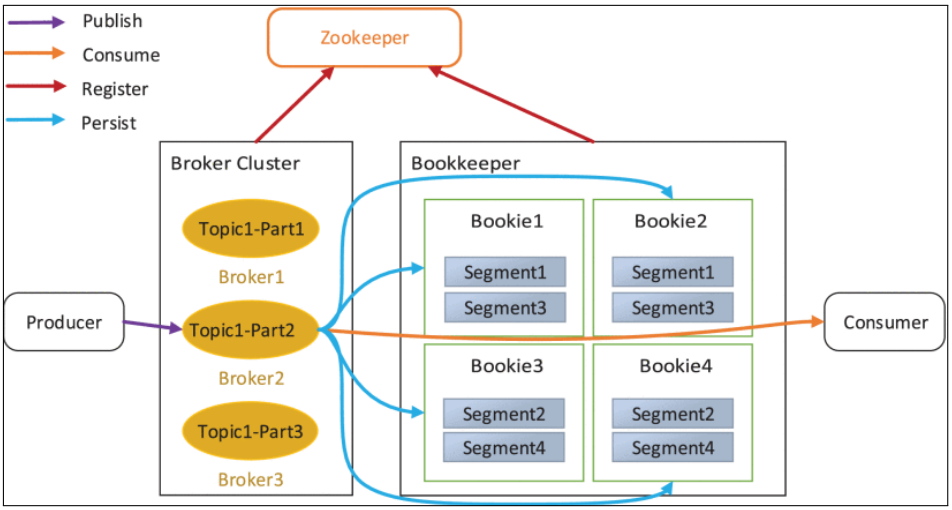


Figure 2. Pulsar's architecture [6].

2.3. Artemis

ActiveMQ Artemis [4] emerges as a robust open-source message queue service, offering a versatile solution for sending, receiving, and storing messages. Its origins in the need for scalable and high-performance messaging systems make it a robust choice for various computing environments, from standalone applications to microservices and cloud-based systems. Developed with modularity in mind, ActiveMQ Artemis, implemented in Java, stands out as a customizable messaging system that supports multiple protocols, including AMQP [9,10] and OpenWire [11]. As a messaging broker, ActiveMQ Artemis facilitates seamless communication between producers and consumers, providing a decoupled approach that ensures efficient information flow. Figure 3 [6] displays the architecture of Artemis.

ActiveMQ Artemis encapsulates a range of features that solidify its standing in the domain of MQs. Its support for various messaging patterns, including point-to-point and publish–subscribe, offers flexibility for diverse communication scenarios. Notably, Artemis introduces advanced messaging features like durable subscriptions, message grouping, and message routing, enhancing the efficiency and customization of message delivery. The platform's clustering capabilities enable the connection of multiple brokers, forming a cohesive and highly available messaging system. This robustness is complemented by ActiveMQ Artemis' support for advanced messaging protocols, contributing to its adaptability in handling intricate communication requirements.

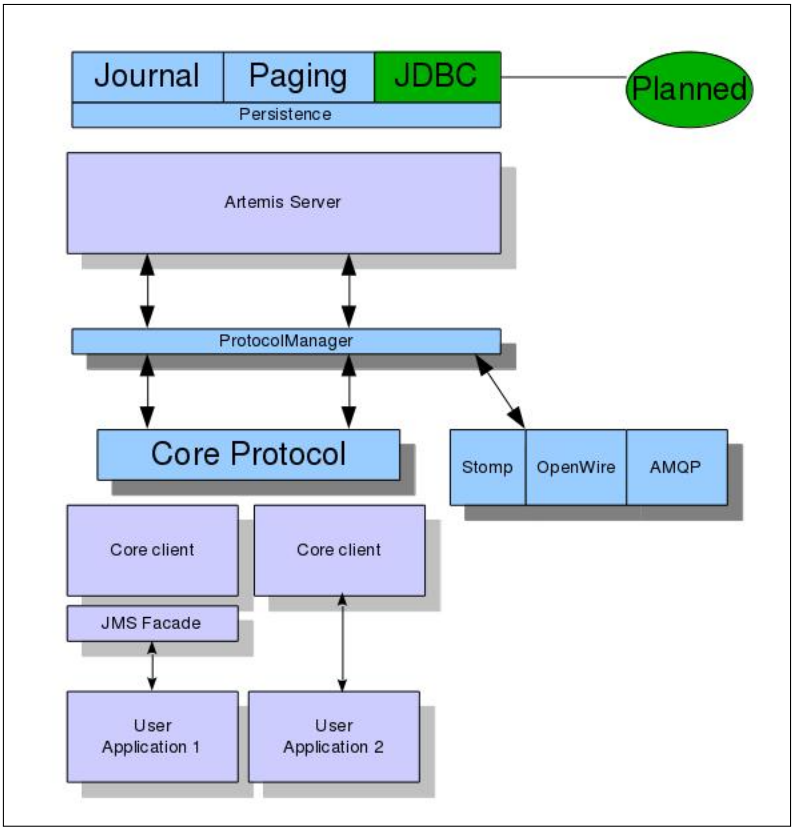


Figure 3. Artemis's architecture [12].

2.4. RocketMQ

Apache RocketMQ [5] is a powerful open-source distributed messaging platform originally conceived to meet the demanding needs of Alibaba's extensive e-commerce ecosystem. It has since evolved into a versatile solution for sending, receiving, and storing messages. It is significantly developed to handle scenarios of high-throughput and fault-tolerant messaging. RocketMQ is being used in applications ranging from large-scale

enterprises to emerging microservices architectures. It offers a robust messaging system that facilitates a seamless exchange of messages, ensuring scalability, reliability, and efficiency.

RocketMQ is a highly resilient and effective messaging system equipped with advanced features and capabilities to handle complex messaging scenarios. It operates on a publish–subscribe model, which enables producers to dispatch records to topics while consumers subscribe to these topics for message consumption. RocketMQ excels in providing durable and scalable messaging, ensuring that messages are reliably stored and can be efficiently processed by consumers. Its support for multiple messaging patterns, including point-to-point and publish–subscribe, and caters to diverse communication needs.

One of RocketMQ's key strengths lies in its ability to handle real-time streaming data. Its architecture, referenced in Figure 4 [6], facilitates the creation of data pipelines, enabling organizations to process and analyze data in real time.

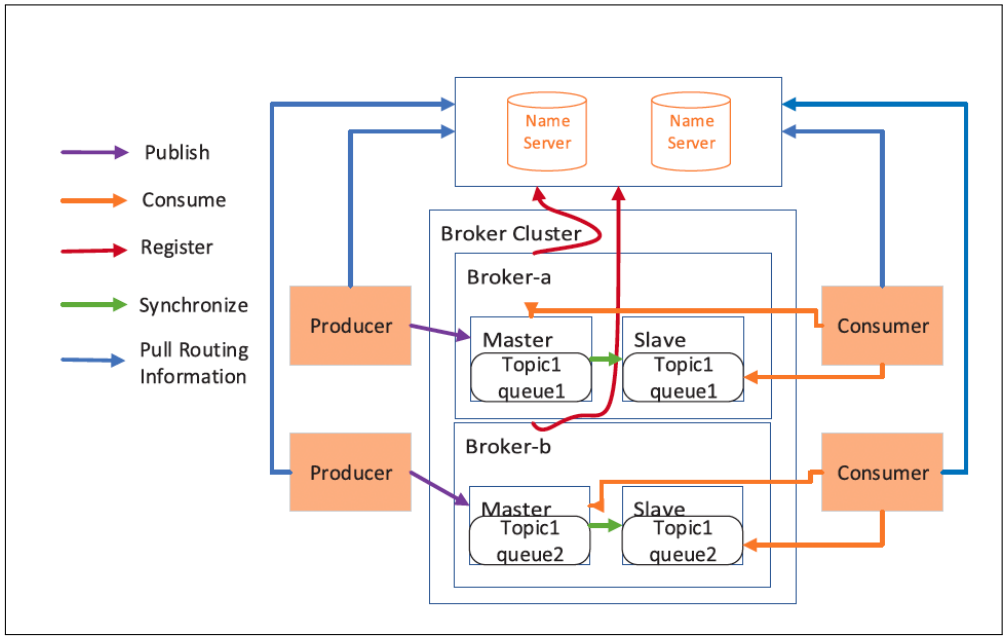


Figure 4. RocketMQ's architecture [6].

3. Related Works

Numerous studies have been conducted on benchmarking message queues, which yield several valuable insights on their performance and characteristics. Piyush Maheshwari and Michael Pang [13] conducted a benchmarking study on comparing two of the message-oriented middlewares (MOMs)—Tibco Rendezvous [14] and Progress Sonic MQ [15]—utilizing SPECjms2007 [16]. Their work focused on testing the effectiveness of message delivery domains, resource utilization, and stability. Similarly, another related study by Ahuja et al. [17] has presented a new benchmark, jms2009-PS, which was built on SPECjms2007 [16] and provided valuable analysis of different MOMs. Another related study by Kai Sachs et al. [18] has introduced a methodology using the SPECjms2007 [16] benchmark, focusing on message traffic analysis and workload customization evaluating the MOM performance. While all these related studies have provided in-depth insight into specific well-known MOMs, our current research differs by presenting a comprehensive and comparative evaluation of JVM-based message queue services: Kafka [2], Artemis [4], Pulsar [3], and RocketMQ [5].

In Fu et al. [6] research study, they compared popular message queue technologies such as Kafka, RabbitMQ [19], RocketMQ [5], ActiveMQ [4], and Apache Pulsar. Their comparison was based on factors like message size, number of producers and consumers, and number of partitions. Although Kafka had high throughput, they noted its latency limitations with larger message sizes. Another study by John et al. [20] focused on com-

paring Apache Kafka and RabbitMQ based on metrics such as throughput and latency. Their studies emphasized that Kafka had superior throughput, while RabbitMQ had better reliability, especially in scenarios prioritizing data security based on the benchmarking tool Flotilla [21]. Our current research work additionally involves a few more JVM-based message queues—Kafka, Artemis, Pulsar, and RocketMQ—and conducts a comparative analysis across diverse workloads by focusing on additional metrics other than latency and throughput. Maharjan et al. [7] conducted an extensive benchmark analysis evaluating the performance of Redis [22], Artemis, RabbitMQ, and Kafka in terms of latency and throughput. Their evaluation highlighted that Redis and Kafka outperformed in latency and throughput, respectively. Additionally, our current research thoroughly evaluates the MQ services of RocketMQ and Pulsar based on critical metrics such as CPU utilization, JVM memory usage, number of threads, and deadlock.

Valeriu Manuel Ionescu et al. [1] research conducted a comparative analysis of RabbitMQ and ActiveMQ, focusing on their publishing and subscribing rates. They compared single and multiple publisher–consumer scenarios using various image sizes to provide a real-world comparison. Sahran Qusay et al. [23] conducted a comparative analysis of Apache ActiveMQ and Apache Apollo [24] message queues, focusing on their messaging capabilities. Their study highlighted that, in most test scenarios, ApacheMQ has outperformed in terms of message-receiving throughputs, and Apache Apollo has outperformed in terms of message-sending throughputs. Our research is an extension of all these studies as we included Pulsar and RocketMQ and analyzed the performance of these message queues under diverse workloads on various metrics. Although their research works highlighted the performance of the two message queues, our current study provides valuable insight into comparative and informed decision-making analysis by evaluating two more JVM-based message queue services in addition to ActiveMQ and Kafka.

Philippe, D. and Kyumars, S.E. [25] introduced a comparison framework to analyze the core functionalities of publish/subscribe systems in both Kafka and RabbitMQ. Their valuable results highlight that replication has a drastic impact on both MQ performances and affects Kakfka more than RabbitMQ. In the same domain, another study conducted by Raje, S [26] compared the performance characteristics of Kafka, Apache ActiveMQ, and RabbitMQ and observed that Kafka outperformed the other two considering factors like message size. Marko et al. [27] conducted a study to compare and contrast the performance of Apache Kafka and RabbitMQ, focusing on flow control and load balancing in IOT cloud-based scenarios. Their study resulted in Kafka having a lower CPU usage and being stable. One of the metrics considered in our current research is CPU usage along with throughput and latency; additionally, we extend the work by comparing Kafka with Artemis, Pulsar, and RocketMQ.

Souza et al. [28] conducted a comparative study between Apache Kafka and RabbitMQ, which extends the work of Philippe, D. [25] and provided the insight that Kafka is more scalable compared to Rabbit MQ in the case of huge throughputs and in some scenarios, with befitting throughput, RabbitMQ outperformed Kafka. Another comparative study conducted by Andrei, K. [29] focused on ActiveMQ and OpenMQ JMS message brokers in terms of message payload size and showed that ActiveMQ was faster than OpenMQ using less memory in all the test scenarios. In our study, we aimed to evaluate the performance profiles of four different JVM-based message queue services, namely Kafka, Artemis, Pulsar, and RocketMQ. Our analysis offers valuable insights into the strengths and limitations of each of these services, allowing for the optimal selection of the host machine based on requirements for CPU utilization, memory, and disk usage. Additionally, our results provide a better understanding of the performance of each message queue under diverse load scenarios.

Table 2 compares our work to the existing literature.

Table 2. Comparison of current research with existing literature.

References	Benchmark	Message Queues				Metrics				
		Kafka	Artemis	Pulsar	Rocket-MQ	Latency	Throu-ghput	CPU Usage	Mem. Usage	Other Metric
Piyush et al. [13]	SPECjms2007									✓
Ahuja et al. [17]	jms2009-PS									✓
Kai Sachs et al. [18]	SPECjms2007			✓					✓	
Fu et al. [6]		✓	✓	✓		✓	✓			
John et al. [20]	Flotilla		✓	✓		✓	✓			
Rokin et al. [7]	OpenMessaging	✓	✓			✓	✓			
Valeriu et al. [1]			✓							✓
Sahran Qusay et al. [23]			✓							✓
Philippe, D et al. [25]		✓								✓
Raje, S [26]		✓	✓			✓	✓			
Marko et al. [27]		✓								✓
Souza et al. [28]	OpenMessaging	✓					✓	✓		✓
Andrei, K. [29]		✓								✓
Our study	OpenMessaging	✓	✓	✓	✓	✓	✓	✓	✓	✓

Note: the checkmark symbol (✓) indicates the inclusion or consideration of additional metrics in the respective studies.

4. Methodology

Different types of industry standards are widely used to benchmark JVM-based systems. Some of those tools are OpenMessaging Benchmark Framework, SpecJMS2007, and Java Microbenchmark Harness (JMH). We have decided to use The OpenMessaging Benchmark Framework for several reasons.

- Able to benchmark several types of message queues.
- Easy to manipulate message size.
- Easy to manipulate message queue configurations like the number of queues, the number of topics, the number of partitions per topic, etc.
- Ability to output experimental results as JSON, making further analysis a breeze.
- Open source and codes can be changed to fit particular needs.

Apart from the above obvious benefits, The OpenMessaging Benchmark Framework has the capacity to generate all the matrices that are required to benchmark a message queue. All the matrices generated by The OpenMessaging Benchmark Framework are discussed below.

In the intricate domain of message queuing, our research endeavor descends into an extensive exploration, meticulously dissecting the technological faculties of Kafka, Pulsar, Artemis, and RocketMQ, with a spectrum of performance metrics as our guide through this enigmatic journey. The metrics, comprising latency, throughput, memory usage, CPU utilization, and garbage collection efficiency, serve not merely as evaluative tools but narrators, each telling its own story of system performance, interweaving tales of efficiency, reliability, and robustness within the complex fabric of message queuing systems.

A deep dive into **latency** reveals a narrative of temporal intervals, speaking to the duration within which messages traverse from producer to consumer. In an environment where time is often synonymous with efficiency, reduced latency illustrates swift message delivery, often equating to an elevated user experience, while its counterpart might hint at possible lags or system delays, potentially impacting real-time communication and transactional systems.

Throughput, conversely, serves as a metric detailing the system's capacity to effectively manage message traffic. It mirrors the volume of messages per unit of time that a

system can judiciously handle, thereby reflecting on its capacity to meet demand or, alternatively, signaling potential points of bottleneck or system strain under voluminous loads.

Peering into the realm of **memory usage**, we extract tales of operational adroitness, unfolding stories of equilibrium or possible discord between system workload and resource allocation, thereby shedding light on whether the system thrives in harmony or potentially teeters on the brink of resource exhaustion.

CPU utilization becomes our window into the efficiency with which computational resources are harnessed, revealing the extent to which the system exploits computational prowess or, perhaps, hints at latent inefficiencies, impacting overall operational productivity. Moreover, the tales told by **garbage collection efficiency** offer insight into memory management within the JVM environment, revealing the proportion of computational time devoted to memory reclaim, which in turn impacts overall system performance, particularly in high-throughput scenarios where resource optimization is paramount.

For our analytical endeavors, we concentrate on these four distinct JVM-based message queues with an astute focus on their behavior within the JVM boundaries. We delve into how each system navigates through the JVM's garbage collection cycle, its memory usage patterns, and its computational resource utilization. Furthermore, we probe into the latency and throughput characteristics, offering a multi-faceted view of each system's performance profile.

To meticulously record data pertaining to the garbage collection cycle, CPU usage, and JVM memory usage, we employ a standard JMX exporter provided by Prometheus [30]. This JMX exporter, deployed as a Java agent, exposes an HTTP server port, serving as a data source for our visualization platform, Grafana [31]. Grafana, in its role, queries the Java agent at pre-defined intervals (every second in our experiment), ensuring data are both current and relevant.

In our pursuit of latency and throughput data, we leverage The OpenMessaging Benchmark Framework, which has etched its place in the industry as a reputable standard for benchmarking message queues. It provides an unbiased estimation of latency and throughput, encapsulating various percentile data. In our context, we delve into the 50th, 75th, and 95th percentile data for latency, complemented by average values, while throughput analysis is approached through evaluating the average rate at which messages are produced and received across nodes, offering a reflective measure of potential bottlenecks within the queue's confines. The methodology's flow diagram is illustrated in Figure 5.

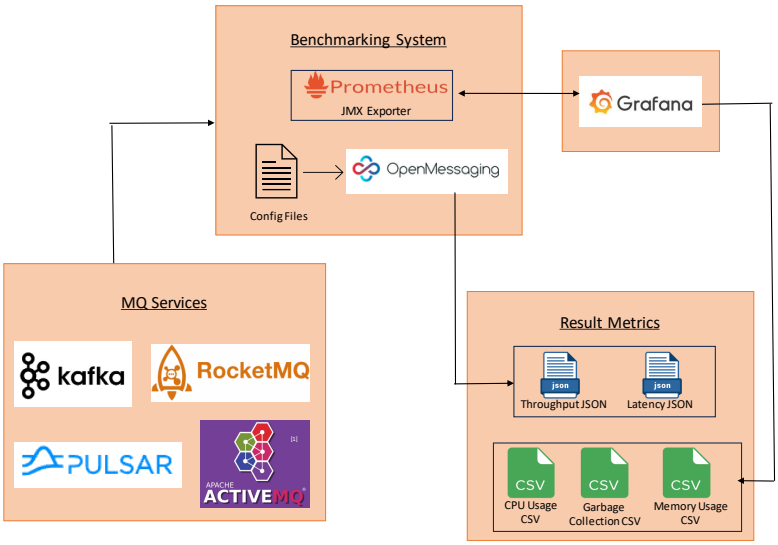


Figure 5. Methodology flow diagram.

While conducting benchmarking with The OpenMessaging Benchmark Framework we kept several parameters unchanged all across the experiments. These parameters are crucial for ensuring unbiasedness towards a particular message queue.

- Same network configuration like dedicated ethernet connection from OpenMessaging Benchmark Framework node to message queue server to ensure minimal latency and eliminate all the variables that can hinder packet passing through the network.
- Identical virtual machine for all the master and subordinate nodes of message queues.
- Identical deployment configuration for all the message queues. All the message queues were deployed in three master and three subordinate nodes to mitigate delay in reaching quorum consensus in case of node failure.
- "One-topic-one-partition" for all the message queues.
- Identical message size for all the message queues.
- While running the benchmark it was made sure that no unnecessary processes were running in the JVM.
- After each experiment, enough time was given to "cool down" the JVM so that the next experiment cycle would not fall into the previous garbage collection cycle.

5. Experiments

In this research, a systematic and unbiased benchmarking of several message queue systems was conducted in a controlled and isolated environment. Virtual machines, each configured with an 8 Core CPU (Intel Core i7), 8 GB RAM, and a 256 GB HDD and Alma Linux 9.1 minimal as host OS, were utilized to host each of the message queue systems, ensuring identical computational and storage resources to obviate variables that could introduce discrepancies in the findings.

For engendering a failsafe during the experimentation phase, every message queue system was deployed in a master–subordinate configuration. This was orchestrated by employing three master nodes and an equal number of subordinate nodes, laying down a structured, resilient architecture that militated against data loss and facilitated an unbroken continuum in the operational environment during the testing phase.

The JMX-exporter was strategically deployed as an independent Java agent within the Java Virtual Machine (JVM), revealing a vital HTTP port that was exposed to facilitate external data querying. This specifically designated HTTP port was, in turn, utilized as a data source for Prometheus, which parsed the JMX matrices into time series data, subsequently visualized through Grafana, thereby providing a nuanced, graphical overview of Kafka's operational dynamics.

The derived metrics from the JMX exporter, which encapsulated percentages of CPU utilization, memory utilization, and garbage collection cycles across all nodes, were subjected to meticulous scrutiny. Following data extraction from all the nodes, an aggregated average was employed within Grafana to calculate and represent the average data, providing a coherent and unified data representation for ease of analysis.

In the realm of latency and throughput measurement, the benchmarking of the message queue systems was conducted by subjecting each message queue to four different load sizes, 1 Kb, 10 Kb, 100 Kb, and 1000 Kb and subsequently gauging the system's ability to manage these loads efficiently. The main reason for choosing those four message sizes was to mimic a real-world problem where loads increase in geometric progression of tenfold. The comprehensive latency analysis focused on end-to-end latency for different percentiles under varying throughput conditions, using specific message sizes to carefully tease out the intricate performance nuances of each MQ system.

In the context of performance metric analysis, notably concerning CPU and memory utilization, as well as garbage collection cycles, a discerning approach was adopted. Specifically, CPU utilization metrics offered insights into the computational demands placed upon the system during the processing of varying message throughputs and sizes. Memory utilization provided a window into the efficiency and efficacy with which each message queue system managed its allocated memory resources under different operational stresses.

Lastly, an exploration of the garbage collection cycles presented a lucid picture of how effectively each system managed memory allocation and deallocation, a critical aspect influencing latency and throughput, especially under high-load scenarios.

It is evident from statistics that one sample size may lead to incoherent and biased results. To avoid bias towards a particular message queue, we conducted each experiment 10 times on the same experimental settings and took their averages to eliminate any discrepancy. It is to be noted that we did not tweak any deployment and message processing settings of any message queue. Because of this presumed notion, we did not explore the behaviors of message queues if subjected to message loads greater than 1000 Kb. If we wanted to analyze their behaviors on higher load size, then we would have to tweak a lot of settings in message processing, which includes batch processing. Deploying batch processing will give unfair advantages to some message queues, notably Kafka. Because of these reasons and to give developers a perspective on how each message queue behaves out of the box, we abstained from tweaking any settings.

When collecting the JVM matrices, we left the JVM to cool down for 10 min after each experiment. Here cooling means restarting the JVM, letting it sit idle to eliminate any chance of interference from residual threads and pre-compiled code in the JIT compiler from the previous experiment. This was carried out to obtain coherent matrices and make sure that an experiment did not influence the next experiment.

6. Results

Our experimental outcomes are diligently partitioned into diverse metrics, encompassing throughput, latency, CPU utilization, memory usage, and garbage collection efficiency, with each metric being allotted its dedicated subsection for meticulous exploration and discussion. Visual aids in the form of graphs facilitate an effortless comparative analysis across the four message queues under scrutiny, while associated tables, situated below each graph, furnish readers with exact numerical data for an in-depth evaluation. Given the potentially vast disparities among various data points, a logarithmic scale has been applied to the y-axis of all throughput and latency. Consistent color assignments have been utilized for each message queue throughout all visual materials to ensure an unambiguous, reader-friendly experience: a specific color denotes the same message queue consistently across all graphical illustrations.

Readers might find it confusing when interpreting the RocketMQ data in case the data size is larger than 1 KB. We have dedicated a separate in-depth analysis in the discussion Section 7.5 on why RocketMQ cannot handle a data size of 10 KB or more with the messaging rate of The OpenMessaging Benchmark Framework. Due to RocketMQ's inability to handle larger message sizes, we set all behavioral matrices to zero..

6.1. Latency

In our journey to dissect and comprehend the latency profiles of Artemis, Kafka, Pulsar, and RocketMQ across distinct message sizes (1 KB, 10 KB, 100 KB, and 1000 KB), each percentile offers us a unique lens through which we can peek into the very heart of each messaging system's performance, which has been depicted in Figures 6–10.

In the context of 1 KB messages, a deep dive into various percentiles reveals Artemis to steadfastly uphold low latency, consistently demonstrating minimal delays from the 50th (median) to the 99.99th percentile, thus illustrating a reliable and robust performance. Conversely, Pulsar showcases a stark contrast, with notably high latency across all percentiles, peaking sharply in the 99th and 99.99th percentiles. This could imply challenges or perhaps intentional design choices in managing small message sizes under differing loads. Kafka and RocketMQ present middle-ground performances, yet RocketMQ uniquely exposes an intriguing latency spike in the higher percentiles, indicating potential bottlenecks or erratic behaviors in specific scenarios.

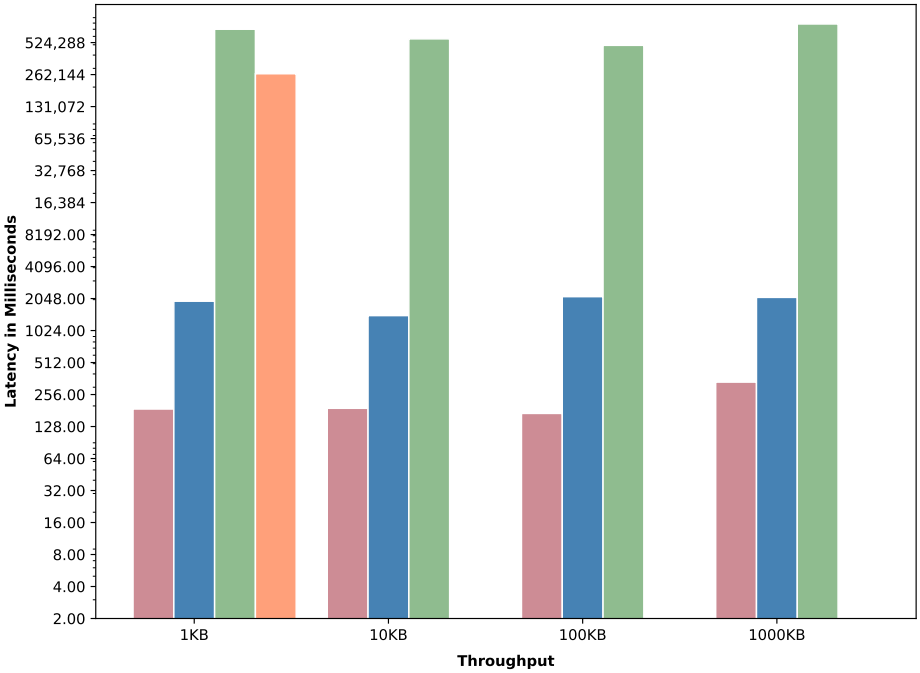
When evaluating the 10 KB message size, Artemis preserves its trend of minimized latency across all percentiles, cementing its capability to efficiently manage messages of

this size. Pulsar, continuing its trend, magnifies its latency figures, especially in the upper percentiles, shedding light on potential scalability or system design considerations. Kafka generally maintains a moderate and somewhat uniform latency across all percentiles, neither excelling nor faltering dramatically in its performance.

Inspecting the 100 KB message size, Artemis begins to exhibit slightly elevated figures in the upper percentiles, hinting at a gradual system strain with increasing message size while still maintaining an admirable performance. Pulsar amplifies its high-latency trajectory, becoming notably pronounced in the higher percentiles, which might be attributed to inherent system design decisions or potential areas for optimization. Kafka remains a steady performer, showcasing a mildly increasing yet stable latency across all percentiles.

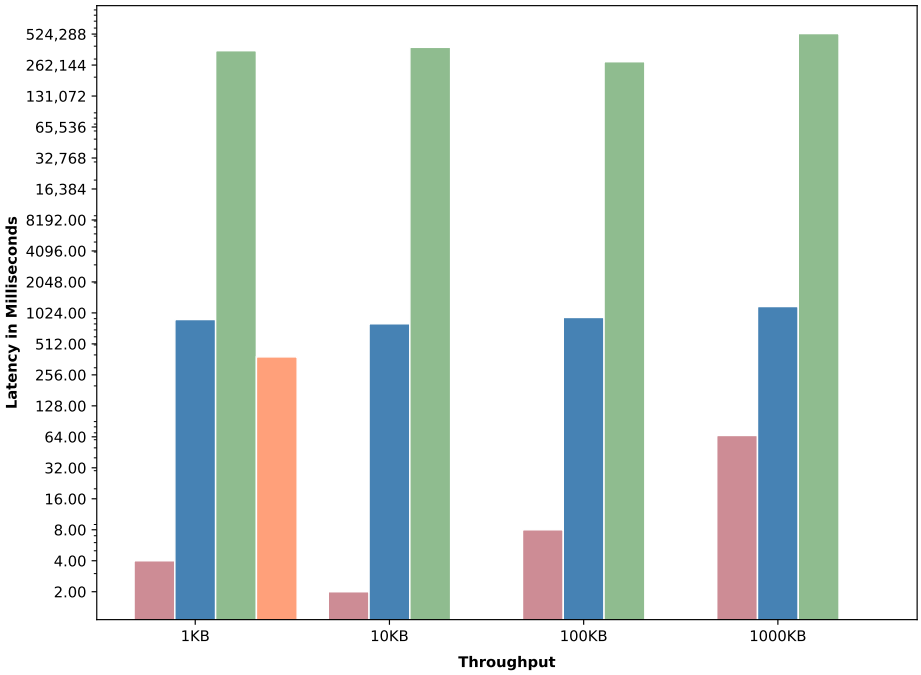
For the 1000 KB messages, the latency analysis elucidates further trends. Artemis, while maintaining relatively low latency, reveals a noticeable incremental pattern as the percentiles ascend, thereby hinting at a system strain under heavier loads with larger messages. Kafka maintains its modest and stable latency, providing a balanced performance even at larger message sizes. Pulsar further escalates its latency figures, particularly in the upper percentiles, indicating potential stress points or deliberate trade-offs in its system architecture and design.

In essence, a meticulous percentile-based analysis underscores Artemis's adept handling of latency across message sizes, Kafka's stable and consistent performance, and Pulsar's progressively escalating latency, which weaves a complex tapestry for subsequent inquiries into the intrinsic architectural and configurational choices, optimization opportunities, and potential trade-offs embedded within these messaging systems. This foundational analysis thus not only enriches our understanding of each system's latency management but also sets the stage for future, in-depth explorations and validations.



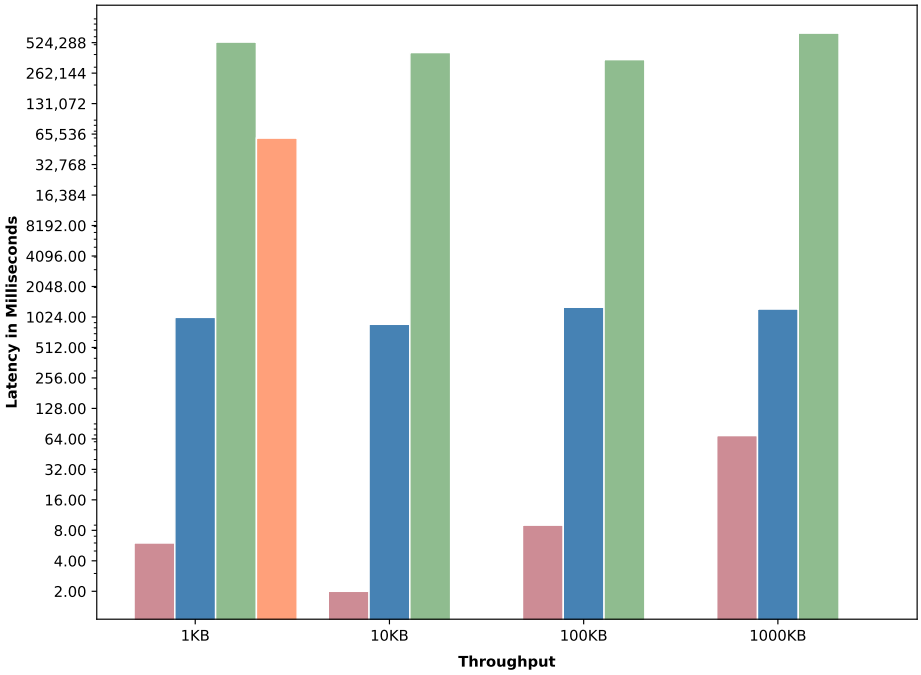
	1KB	10KB	100KB	1000KB
Artemis	187.0	190.0	170.0	335.0
Kafka	1933.01	1415.01	2133.01	2100.01
Pulsar	700,014	568,283	493,887	784,666
RocketMQ	266,924	0	0	0

Figure 6. Maximum end-to-end latency. Note: the RocketMQ drop for larger data is explained in Section 7.5.



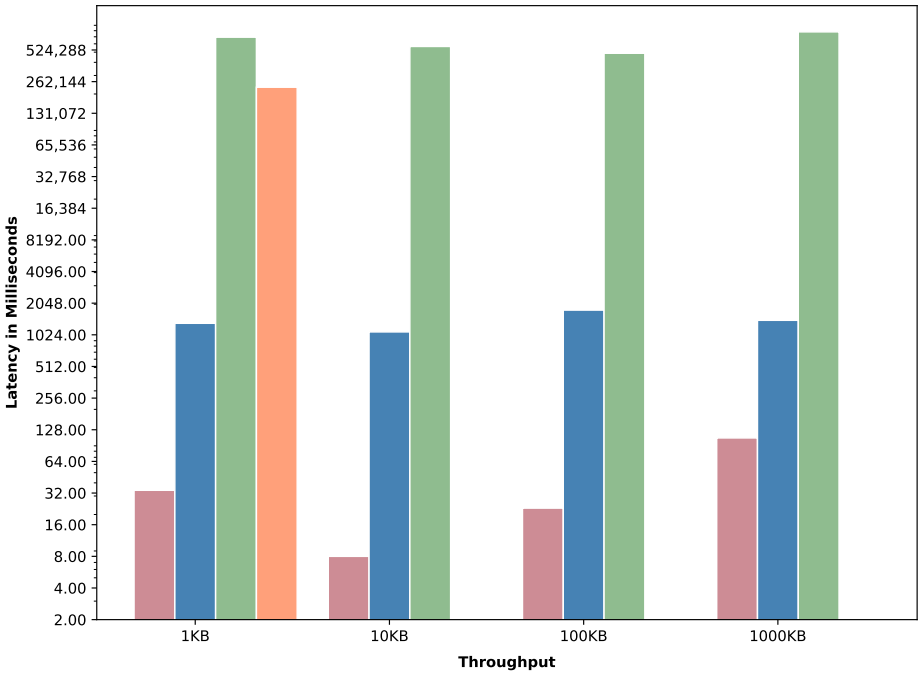
	1KB	10KB	100KB	1000KB
Artemis	4.0	2.0	8.0	66.0
Kafka	883.0	803.0	925.0	1181.01
Pulsar	361,605	389,928	282,611	530,505
RocketMQ	383.0	0	0	0

Figure 7. The 50th percentile end-to-end latency.



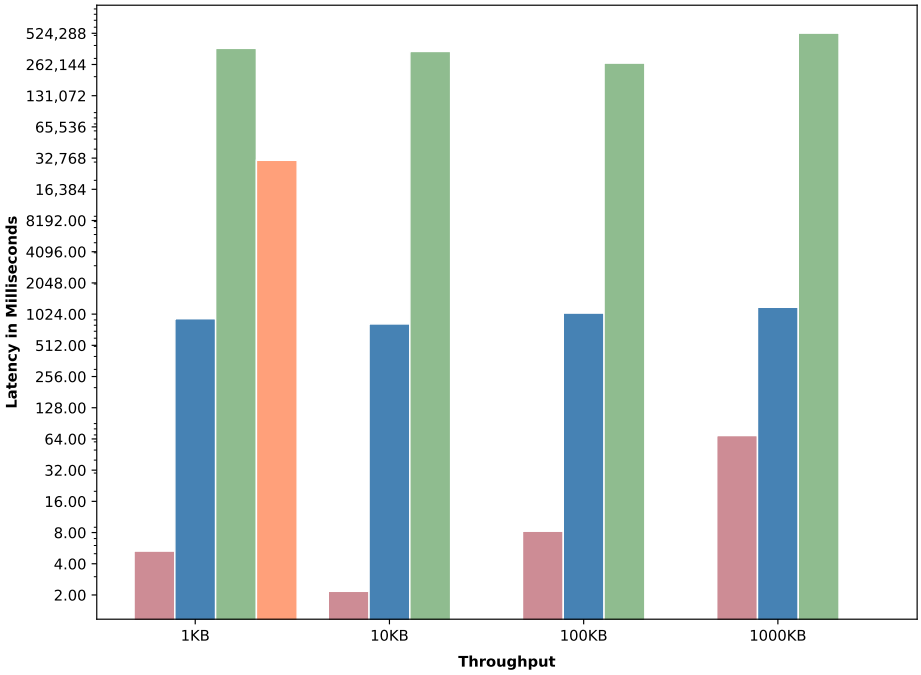
	1KB	10KB	100KB	1000KB
Artemis	6.0	2.0	9.0	69.0
Kafka	1015.0	867.0	1277.01	1226.01
Pulsar	530,900	418,381	356,354	651,091
RocketMQ	59,812	0	0	0

Figure 8. The 75th percentile end-to-end latency.



	1KB	10KB	100KB	1000KB
Artemis	34.0	8.0	23.0	107.0
Kafka	1318.01	1091.01	1759.01	1407.01
Pulsar	694,026	566,358	488,103	779,649
RocketMQ	231,954	0	0	0

Figure 9. The 99th percentile end-to-end latency.

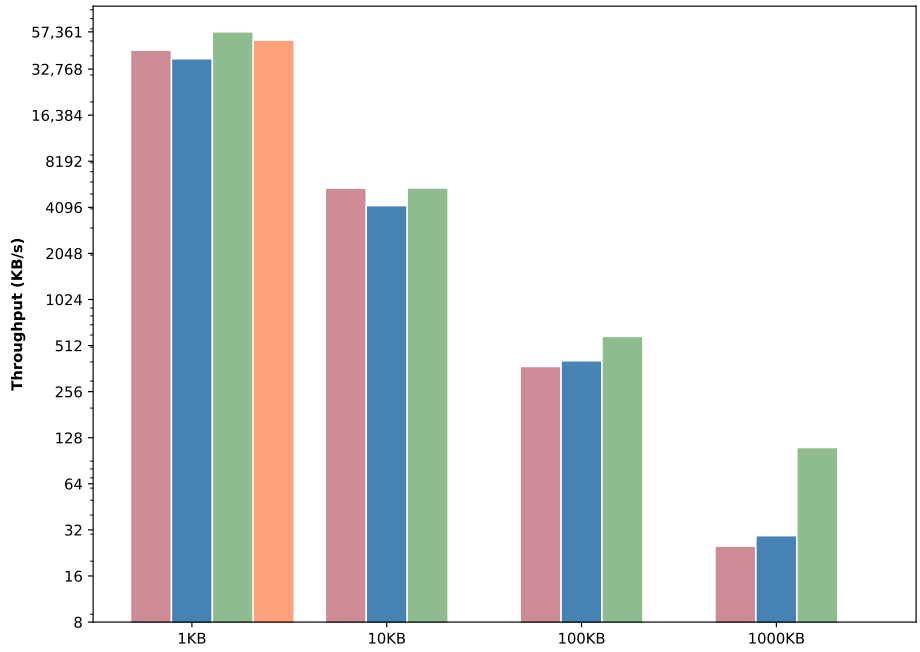


	1KB	10KB	100KB	1000KB
Artemis	5.3	2.17	8.24	69.02
Kafka	924.55	823.47	1048.12	1192.3
Pulsar	375,243	350,261	270,300	526,213
RocketMQ	31,221	0	0	0

Figure 10. Average end-to-end latency.

6.2. Throughput

In the evaluation of maximum throughput across distinctive load sizes, our experimental results illuminated a compelling narrative about the performance of Kafka, Artemis, RocketMQ, and Pulsar. For smaller message sizes, particularly at 1 KB, Pulsar demonstrably outperformed its counterparts, attaining the highest throughput among the quartet. However, as the message size expanded, discernable variations in performance were observed in Figure 11. The resilience and robustness of Pulsar were particularly noticeable, consistently maintaining the highest throughput across all explored message sizes. Surprisingly, RocketMQ, despite demonstrating a competitive throughput at smaller message sizes, did not exhibit results in larger payloads, raising intriguing questions about its performance scalability and robustness amid escalating load sizes. Kafka and Artemis, while trailing behind Pulsar, exhibited a commendable degree of consistency across varying message sizes, although with diminishing throughput figures in larger load scenarios.



	1KB	10KB	100KB	1000KB
Artemis	43,534	5470.77	373.77	25.07
Kafka	38,264	4202.02	407.32	29.33
Pulsar	57,361	5479.77	587.57	110.45
RocketMQ	50,682	0	0	0

Figure 11. Maximum throughput.

6.3. CPU Utilization

Spanning Figures 12–15, a contemplative exploration into the CPU utilization amongst Kafka, Pulsar, Artemis, and RocketMQ unfolds, each system carving its unique resource usage signature upon different message load sizes of 1 KB, 10 KB, 100 KB, and 1000 KB.

In the context of the 1 KB message size, Kafka surfaces as a pronounced consumer of CPU resources, indicating an architecture perhaps optimized for robustness over computational economy. In stark contrast, Pulsar epitomizes efficiency, utilizing the least CPU resources and thereby raising queries about the internal architectural subtleties that afford such frugality. Artemis and RocketMQ, exhibiting moderate CPU utilization, weave a narrative of balanced computational employment, comfortably oscillating between the extremes defined by Kafka and Pulsar.

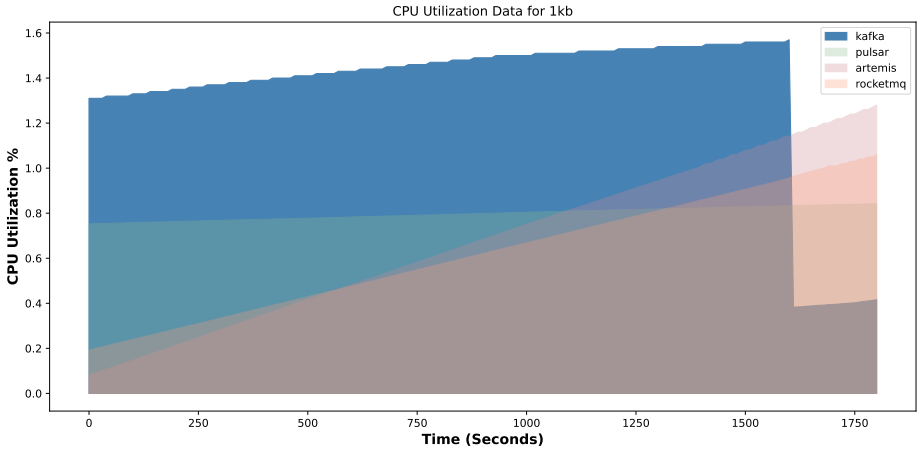


Figure 12. CPU utilization—1 KB.

The 10 KB load size pivots the narrative slightly. Artemis, now emerging as the most resource-intensive option, redirects our attention toward understanding the intricacies of its operational dynamics that necessitate such augmented CPU employment. Kafka retreats towards a more reserved CPU usage, bringing itself closer to Pulsar, which unflinchingly continues its low-CPU-utilization journey. The conspicuous absence of RocketMQ from this juncture forward, due to its inability to manage these larger load sizes, inevitably forms a riveting subplot that beckons deeper investigative probing.

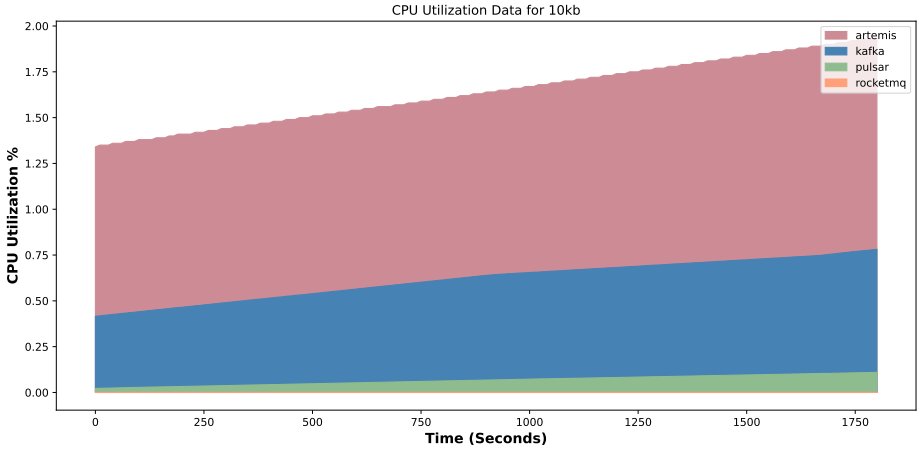


Figure 13. CPU utilization—10 KB.

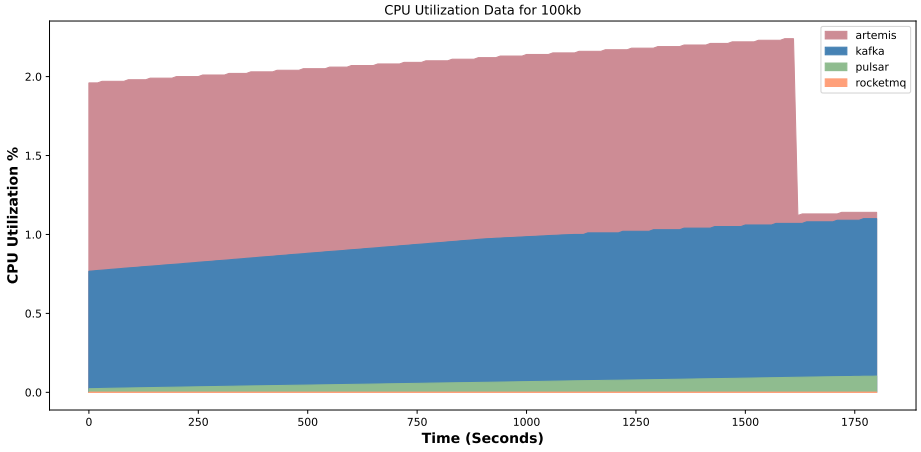


Figure 14. CPU utilization—100 KB.

Amplifying the message load to 100 KB perpetuates the established trends from the 10 KB scenario, with both Artemis and Kafka maintaining their respective CPU utilization characters. This consistency across different message sizes unearths avenues for investigating how varied message loads influence—or perhaps do not influence—resource allocation strategies within these systems.

Navigating to the domain of the 1000 KB message size, Kafka reclaims its position as the preeminent CPU user, evoking thoughts on its scalability and performance strategies, particularly in contexts demanding the processing of substantial message sizes. Artemis moderates its computational consumption slightly, whereas Pulsar remains an exemplar of minimalistic CPU usage, steadfastly sustaining its remarkably low usage across the entirety of the explored message sizes.

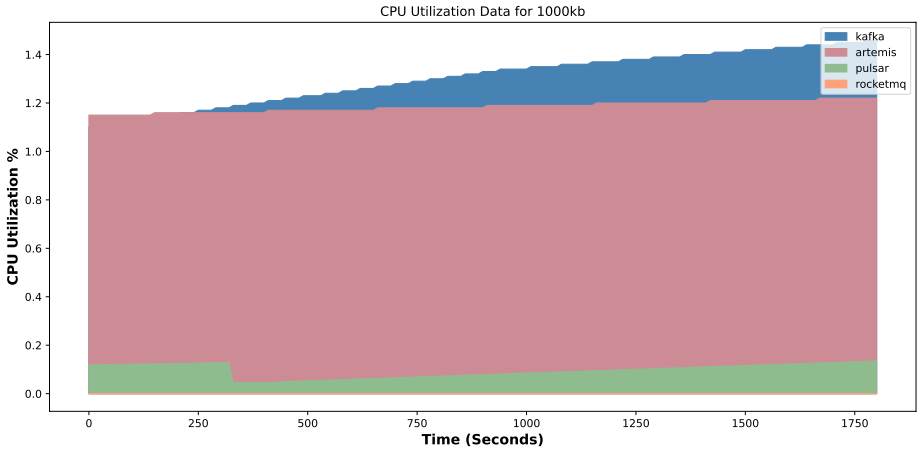


Figure 15. CPU utilization—1000 KB.

6.4. Memory Usage

Figures 16–19 casts an illuminating light on the memory usage dynamics across Kafka, Artemis, Pulsar, and RocketMQ, with differing message payloads, allowing for an analytical examination of how these message queuing systems navigate resource allocation.

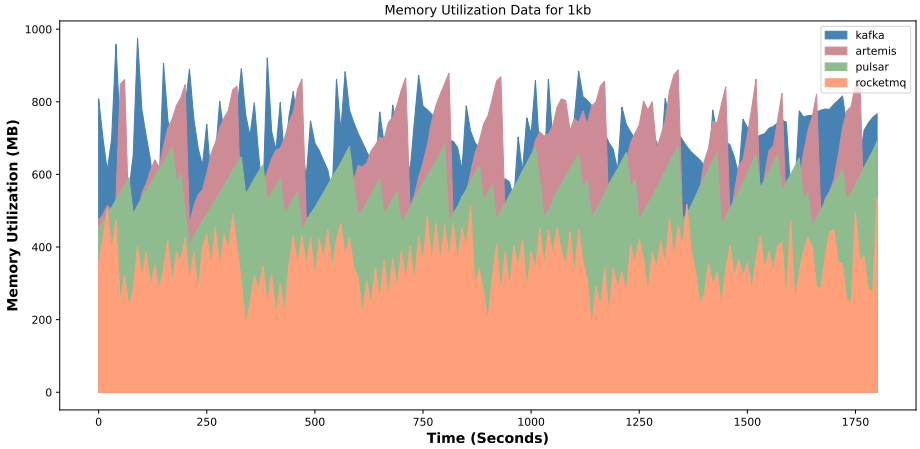


Figure 16. Memory usage—1 KB.

Delving into the 1 KB load size, RocketMQ sets a noteworthy benchmark, exhibiting the most parsimonious memory usage among its contemporaries. Interestingly, while Kafka and Artemis tread along adjacent memory usage trajectories, a closer examination reveals Kafka's nuanced propensity to allocate marginally more memory on average, potentially inferring a slight divergence in their memory management approaches or resource demands for small message handling. Pulsar, residing between RocketMQ's thriftiness and the

relative liberality of Kafka and Artemis, is observed to execute an efficient, medium-usage memory strategy.

As the payload escalates to 10 KB, the absence of RocketMQ underlines a pivotal operational or configurational restraint in handling larger message sizes. This message size presents an analytical curiosity: Artemis punctuates its memory usage with intermittent peaks, revealing a dynamic, possibly adaptive, memory management style, while Kafka sustains a more equitably high memory usage. Pulsar's initial memory surge, followed by a gradual decline and stabilization as the lowest memory user, hints at a possible pre-allocation and subsequent optimization strategy that would benefit from a deeper technological dissection.

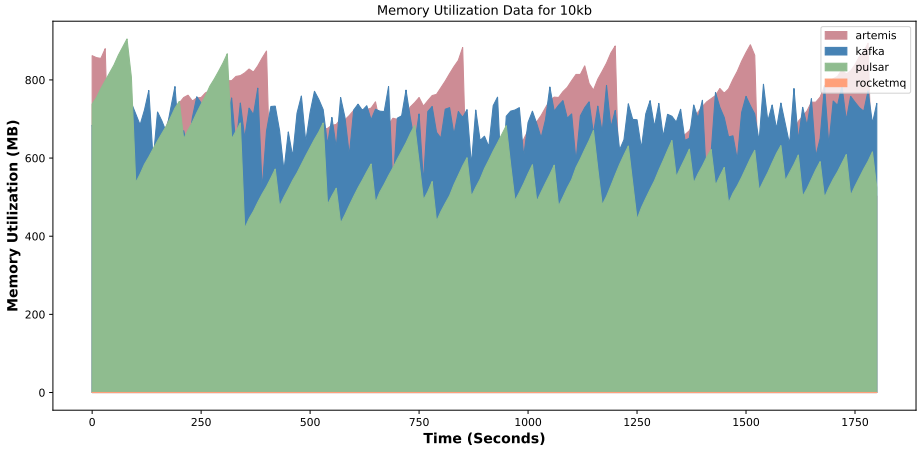


Figure 17. Memory usage—10 KB.

The 100 KB payload mirrors several aspects observed in the 10 KB context, albeit with a few tweaks worth analytical pondering. The initial high memory use by Pulsar prompts questions regarding its setup or warm-up phase, signifying an area that may potentially be streamlined for improved initial resource use. Meanwhile, Artemis and Kafka maintain their consistently competitive, elevated memory utilization, subtly flagging an area that might be optimized or alternatively, might be a considered trade-off for achieving other performance merits.

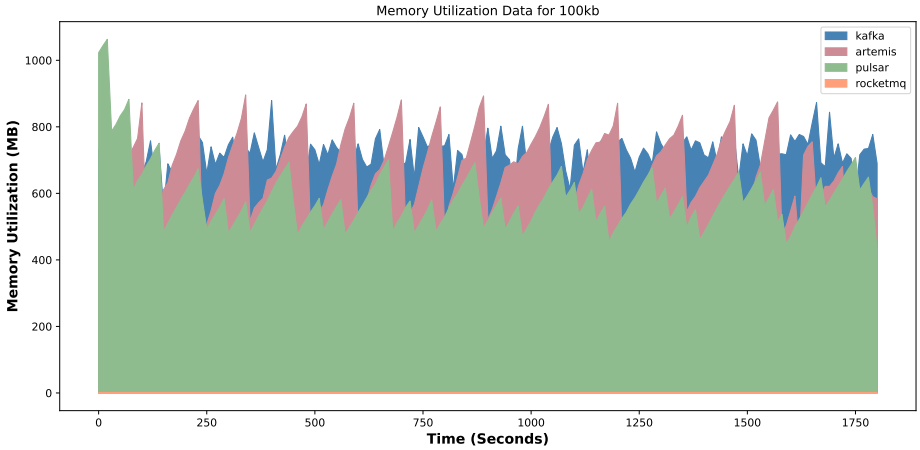


Figure 18. Memory usage—100 KB.

When subjected to the hefty 1000 KB message size, Kafka takes a definitive leap to the forefront in memory usage, a position which, while possibly necessitated by its architectural or functional provisions, warrants a meticulous investigation into its memory allocation and management algorithms. Artemis, while trailing, provides a stable, albeit high memory usage footprint. Pulsar emerges as an embodiment of memory efficiency in this scenario,

adhering to its established trend of minimalistic memory usage and providing a platform for investigating how such memory efficacy is achieved among larger payloads.

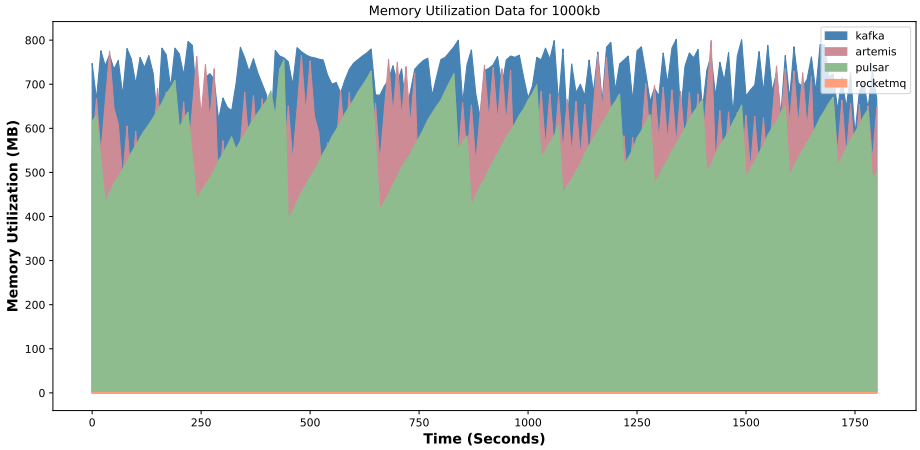


Figure 19. Memory usage—1000 KB.

6.5. Garbage Collection

The examination of garbage collection (GC) in our analysis, as illustrated in Figures 20–23, presents a tableau of operational efficacies and subtle idiosyncrasies exhibited by the selected MQs under differing message loads.

For the 1 KB message load size, Pulsar stands out with the most efficient garbage collection performance, implying robust memory management strategies for small message footprints. Kafka—while not trailing too far behind—exhibits the least optimized GC cycles among the three, suggesting possible challenges or trade-offs in handling smaller messages. Artemis and RocketMQ lie within this range, marking comparable efficiencies.

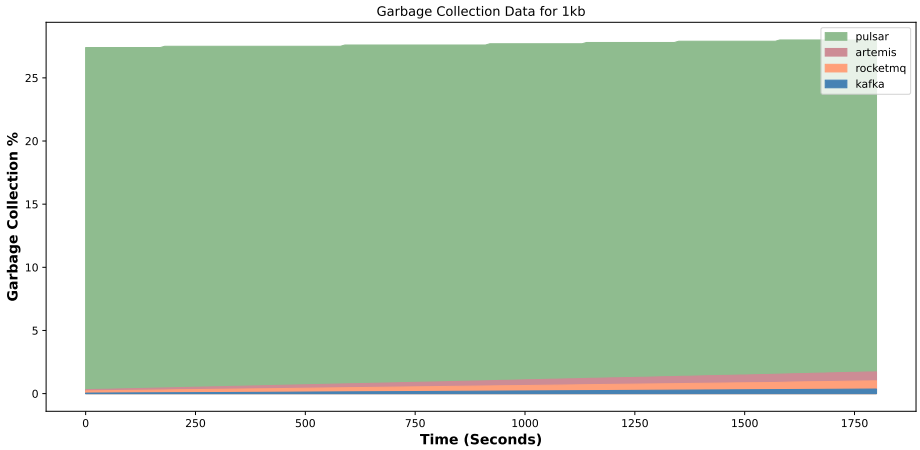


Figure 20. Garbage collection—1 KB.

When the message size is escalated to 10 KB, a distinct shift in performance surfaces. Artemis emerges as the leader in terms of garbage collection efficiency, minimizing its GC interventions and hinting at adaptability in its memory management techniques for this payload scale. Kafka and Pulsar depict a reversal of roles: Pulsar now registers more frequent garbage collection events, perhaps revealing a challenge in its architecture or configuration for this specific message size, while Kafka draws closer to the lower end, showcasing better optimization than the 1 KB case.

The narrative continues at the 100 KB message load size. Artemis sustains its commendable garbage collection efficiency, asserting its supremacy in optimizing memory management

across diverse message sizes. Kafka aligns closer to the median, with Pulsar settling as the least efficient, reinforcing the notion of its potential struggles with larger payloads.

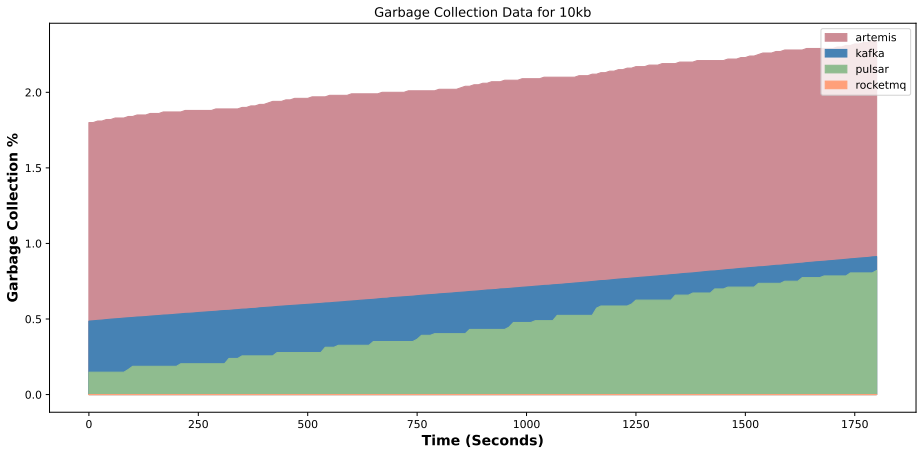


Figure 21. Garbage collection—10 KB.

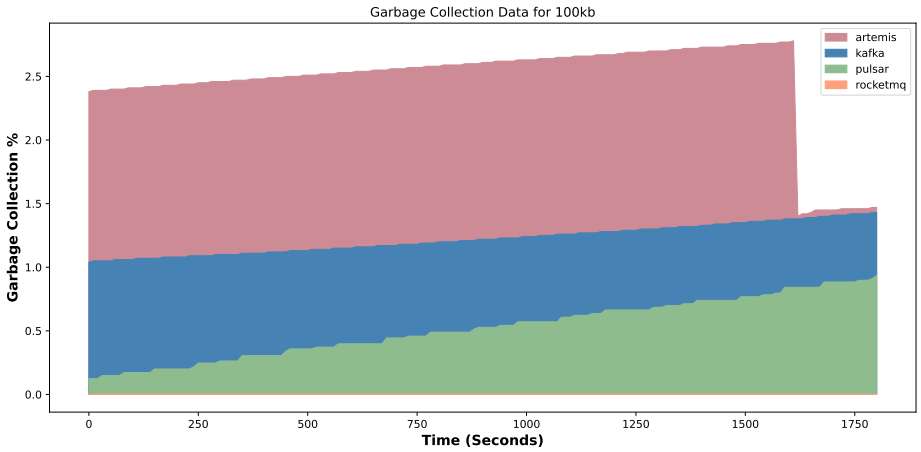


Figure 22. Garbage collection—100 KB.

Remarkably, the 1000 KB data load size inverts the observed trends. Kafka evolved as the front runner in minimizing garbage collection cycles, indicating adeptness in handling considerably larger messages. Artemis retains a proficient stance, situating itself marginally behind Kafka, whereas Pulsar finds itself positioned as the least efficient among the trio.

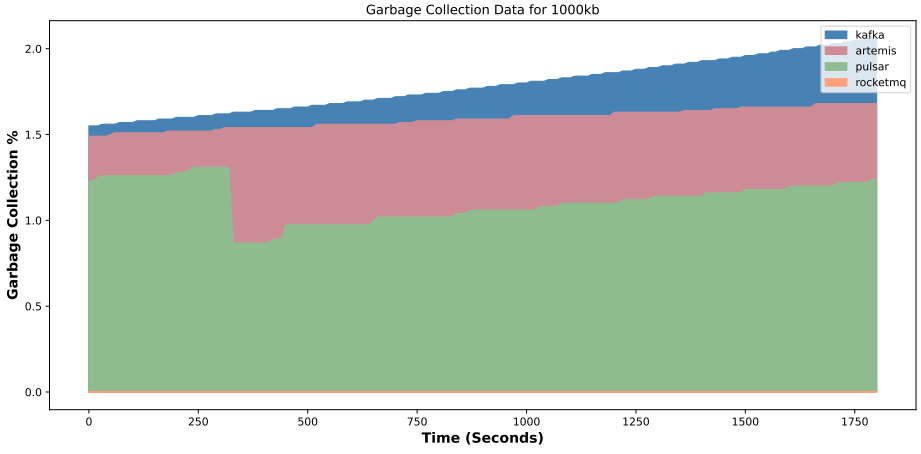


Figure 23. Garbage collection—1000 KB.

7. Discussion

The empirical findings from our experimentation process illuminate intriguing aspects of the four message queue systems in context—Kafka, Pulsar, Artemis, and RocketMQ. Across multiple metrics—namely throughput, latency, CPU utilization, memory usage, and garbage collection—notable distinctions and systemic characteristics were unveiled, informing a nuanced understanding of each platform's operational demeanor and potential applicability in varied use-case scenarios.

7.1. Throughput and Latency: Finding Balance

Reflecting upon throughput and latency inherently necessitates contemplating the trade-offs between delivering messages expediently (throughput) and ensuring messages are processed with minimal delay (latency). On observation, Artemis prominently showcased its finesse in handling smaller message sizes, delivering commendable low-latency performance while still maintaining admirable throughput. Kafka, whilst slightly lagging in the 50th percentile latency, notably exhibited a consistent behavior across evaluated throughputs, signifying stability that might be preferable in certain use-cases where predictable performance is paramount.

Conversely, Pulsar's elevated latency, especially in the upper percentiles, warrants a deeper dive into its architectural choices, while a seemingly compromising position, it is plausible that Pulsar is optimized for scenarios where high throughput is prioritized over minimized latency.

7.2. CPU Utilization: Performance versus Resource Efficacy

The exploration of CPU utilization unveiled variable efficiencies among the tested systems. The distinctive elevation of Kafka's CPU usage, particularly with larger messages, might suggest an inclination toward prioritizing processing power to maximize throughput and minimize latency, possibly at the expense of resource efficiency. Conversely, Pulsar's consistently lower CPU utilization could indicate a trade-off, wherein it economizes on resources but may sacrifice some degree of performance, a theory corroborated by its latency behavior.

Artemis and RocketMQ, providing a middle ground, exhibit variances in CPU usage dependent upon message size, necessitating further exploration into how adaptive CPU utilization strategies might be engineered within these systems to adjust dynamically to varying message loads.

7.3. Memory Utilization: Stability versus Peaks

Memory utilization unveiled a stratification among the systems, with Kafka and Artemis generally consuming higher memory, particularly with escalating message sizes. This might signify a buffering or caching strategy to augment message handling efficiency, thereby influencing throughput and latency.

Pulsar's generally lower memory footprint, save for its initial bursts, may suggest a more controlled memory allocation strategy. Moreover, Artemis's occasional peaks in memory usage, despite its commendable latency performance, hint towards a possible relationship between memory usage and minimized latency, although this warrants further analysis to elucidate any causal relationships.

7.4. Garbage Collection: Minimizing Operational Interruptions

Pulsar's variable performance across different message sizes in garbage collection cycles offers insights into its Java Virtual Machine (JVM) heap management strategies. Whereas efficient in smaller messages, potential inefficiencies materialize with larger payloads, indicating room for heap management optimizations.

Kafka's reduced GC activities, especially with larger messages, suggest a proficiency in managing larger heap sizes, thereby potentially minimizing GC-induced pauses and resultant latency. Artemis's varied garbage collection frequency across message sizes indi-

cates a non-linear relationship between message size and memory management, providing fertile ground for further exploration and potential optimization in future iterations.

7.5. Perspective of Individual Message Queue Systems

In synthesizing the insights from each metric, it becomes patently apparent that each message queue system perhaps embodies a set of trade-offs, wherein optimizations in one metric might invoke concessions in another. Kafka's generally commendable throughput and latency come seemingly at the cost of elevated resource utilization (both CPU and memory).

Artemis, while offering compelling throughput and latency, particularly with smaller messages, reveals potential volatility in memory utilization that warrants scrutiny. Pulsar, whilst conservative in resource utilization, may present compromises in latency and garbage collection efficiency, especially with larger messages.

It is pivotal to acknowledge that architectural, design, and operational choices embedded within each message queuing system might be inherently optimized for specific use-case scenarios. Hence, the judicious selection of a system might well be informed by aligning the intrinsic behaviors and trade-offs of a message queuing system with the contextual demands and tolerances of a given application or use case.

When interpreting our latency data, several careful considerations should be given. In all the latency graphs, it seems that Apache ActiveMQ Artemis has a clear edge over all other message queues. This somehow gives a perceived notation that ActiveMQ Artemis has very low latency and thus can handle very high traffic without any significant bottleneck. However, this finding should be taken with a pinch of salt; this is because, when deploying ActiveMQ Artemis, we have never attached it to any on-disk persistence media. This reduces the huge burden of writing journal and page files from ActiveMQ Artemis's shoulder. On the other hand, Kafka and Apache Pulsar always use on-disk persistence for writing real-time journals and pages. This leads to a significant rise in latency on both Kafka and Apache Pulsar.

Furthermore, readers might find it somewhat confusing when interpreting RocketMQ's behavioral matrices. If the message size is 10 Kb or larger all the matrices are zero because we were not able to make the RocketMQ work out of the box. We have contacted the developers of RocketMQ [32] about the potential issues. As per the discussion, RocketMQ is not able to handle large message sizes with a very high messaging rate present in the OpenMessaging Benchmark Framework. If we want to make the RocketMQ work with higher message sizes, we need to tweak several settings and turn on back-pressure features. However, turning on these features will give RocketMQ unfair advantages compared to other message queues.

When we were conducting the experiments, we observed that if the message size is 10 Kb or larger JVM takes up almost 100% of the host memory. We have tried multiple approaches like deploying the cluster with 8 GB of initial JVM memory and increasing the host machine memory but none worked. Eventually failing at warm-up traffic of the OpenMessaging Benchmark Framework. Our primary guess is that RocketMQ is not able to process all the messages in its queue and its processor threads and queue size keep increasing and eventually taking up all the host memory. When new messages arrive in the queue, it can no longer hold the pressure due to resource starvation and simply gives up to avoid catastrophe. Furthermore, there is a huge bottleneck in the queue and RocketMQ tries to clear the bottleneck but before it can do so new messages arrive in the queue. Ultimately RocketMQ's processors become stuck in an infinite loop.

8. Insights from Our Findings vs. Previous Studies

In prior research examining various message queue systems, throughput, a key metric, has consistently highlighted Kafka's [6,20**?**] and ApacheMQ's [23] superiority considering message receiving. However, our findings echo this trend for Kafka, while revealing limitations in RocketMQ's throughput. In our experiments, Artemis emerged as a latency

front runner, particularly under lighter loads, with Kafka and Pulsar closely following with reliable latency profiles. CPU usage comparisons align with previous research [27], designating Kafka as more resource-intensive, while Pulsar offers a balanced option with a moderate usage. Memory considerations indicate that Kafka and Artemis outperform RocketMQ and Pulsar, showcasing their efficiencies in memory utilization. This depth of insight surpasses the existing research, contributing to a richer comprehension of the strengths and limitations of the chosen message queue systems for our study, thereby facilitating more informed decision making for varied use-case scenarios.

9. Threats to Validity

In our endeavor to comprehensively benchmark the performance of Kafka, Pulsar, Artemis, and RocketMQ, and the associated performance metrics guiding our exploration, it is essential to consider potential threats to the validity of our findings. These threats encompass various aspects of our research methodology, the inherent complexities of benchmarking distributed systems, and the unique characteristics of message queuing technologies. We categorize these threats into four dimensions: construct validity, internal validity, external validity, and conclusion validity.

9.1. Construct Validity

Construct validity pertains to the extent to which our chosen performance metrics accurately capture the essential aspects of message queuing system performance, while we have carefully selected a comprehensive set of metrics, there may be other critical dimensions of system performance that we have not considered. Additionally, the interpretation and relevance of these metrics may evolve with changing technology and system architectures.

To mitigate this threat, we have grounded our choice of metrics in established industry standards and best practices for evaluating message queuing systems. We have also provided detailed explanations of the significance of each metric in assessing system performance. However, it is essential to remain vigilant about emerging trends and evolving performance considerations in the field of distributed systems.

9.2. Internal Validity

Internal validity centers on the robustness of our research methods and the potential for biases or errors. In the context of our study, there are several factors that may introduce biases or limitations. For example, the performance of message queuing systems can be influenced by various external factors, such as network conditions, hardware configurations, and workload patterns. These external influences may introduce variability in our results.

Additionally, the choice of specific configurations and settings for each message queuing system can impact the results, while we strove for consistency and fairness in our benchmarking approach, there may be factors that we have not fully controlled for, potentially affecting the internal validity of our findings.

9.3. External Validity

External validity concerns the extent to which our research findings can be generalized beyond the specific context of our study. Our research primarily focuses on benchmarking Kafka, Pulsar, Artemis, and RocketMQ within the confines of our experimental setup. These findings may not fully represent the diversity of message queuing systems and operational environments found in real-world scenarios.

Furthermore, the performance characteristics of message queuing systems can vary significantly depending on factors such as system scale, data volume, and specific usage patterns. Our findings may provide valuable insights but should be considered within the context of the specific scenarios and configurations we have examined.

9.4. Conclusion Validity

Conclusion validity revolves around the accuracy and reliability of the conclusions drawn from our benchmarking data, while we have diligently designed and executed our benchmarking methodology, it is important to emphasize that our study aims to provide valuable insights and implications rather than claim statistically significant conclusions. The limited scope of our benchmarking scenarios may impact the generalizability of our findings to broader contexts.

Hence, while our research methodology has been rigorously designed and executed, these threats to validity highlight the need for caution in interpreting and applying our benchmarking results. Researchers and practitioners should consider the specific context and limitations of our study when applying our findings to their own message queuing systems and operational scenarios.

10. Conclusions

This study has embarked on an in-depth comparative evaluation of four prominent JVM-based message queue (MQ) services: Kafka, Pulsar, Artemis, and RocketMQ. Our analysis centered around key performance metrics like latency, throughput, CPU utilization, memory usage, and garbage collection efficiency, revealing insights that are critical for their application in distributed systems.

In terms of latency, Artemis demonstrated superior performance, especially under lighter loads, making it an optimal choice for applications where low latency is crucial. Kafka and Pulsar also exhibited reliable latency profiles but are more suited to scenarios where latency is a consideration among other factors. Regarding throughput, Kafka clearly stood out, showcasing its capability to handle large volumes of messages effectively. This positions Kafka as a strong candidate for high-throughput requirements. Conversely, RocketMQ showed certain limitations, suggesting its applicability in less demanding scenarios in terms of message volume.

CPU utilization was another critical area of our study. Kafka's significant computational demand makes it more suitable for environments with ample computational resources. Pulsar, with its more moderate CPU usage, emerged as a balanced option, potentially fitting a wider range of use-cases. Memory usage patterns revealed that Kafka and Artemis, while powerful, would be best deployed in environments where memory resources are not a limiting factor. On the other hand, RocketMQ and Pulsar, with their efficient memory utilization, are appealing choices for systems where memory conservation is essential. The study also delved into garbage collection performance, highlighting the varied efficiency of these MQ services under different operational scenarios. This aspect underlines the importance of considering the specific requirements of a system, especially when garbage collection can impact overall performance.

The overarching conclusion from our research is that the selection of an MQ service should be a nuanced decision, guided by a thorough understanding of each service's strengths and limitations in relation to the intended application context. The optimal choice varies significantly based on specific performance requirements and resource constraints.

Future research should delve into additional performance metrics such as Disk I/O and Network I/O, which are crucial for a more comprehensive understanding of these systems under varied conditions. Exploring the scalability, security, and interoperability of MQ services will also be pivotal, particularly in the context of evolving cloud-based and decentralized computing paradigms. Such investigations will not only enhance our understanding of these systems but also contribute to the development of more robust, adaptable, and secure distributed messaging frameworks.

Author Contributions: M.S.H.C.: resources, conceptualization, methodology, software, data curation, formal analysis and investigation, and writing—original draft preparation. M.A.R.A.: resources, data curation, validation, and writing—original draft preparation. S.M.T.: formal analysis and investigation, validation, and writing—original draft preparation. T.C.: formal analysis and investigation,

supervision, validation, and writing—original draft preparation. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Informed Consent Statement: Not applicable.

Data Availability Statement: The results in JSON and CSV format can be found under *results* folder of the GitHub repository https://github.com/showkat2203/benchmarking-message-queues (accessed on 25 November 2023).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ionescu, V.M. The analysis of the performance of RabbitMQ and ActiveMQ. In Proceedings of the 2015 14th RoEduNet International Conference—Networking in Education and Research (RoEduNet NER), Craiova, Romania, 24–26 September 2015; pp. 132–137. [CrossRef]
2. Apache Software Foundation. Apache Kafka. Available online: https://kafka.apache.org/ (accessed on 12 October 2023).
3. Apache Software Foundation. Apache Pulsar. Available online: https://pulsar.apache.org/ (accessed on 11 January 2023).
4. Apache Software Foundation. ActiveMQ Artemis. Available online: https://activemq.apache.org/components/artemis/ (accessed on 31 January 2023).
5. Apache Software Foundation. Apache Rocketmq. Available online: https://rocketmq.apache.org/ (accessed on 12 October 2023).
6. Fu, G.; Zhang, Y.; Yu, G. A fair comparison of message queuing systems. *IEEE Access* **2020**, *9*, 421–432. [CrossRef]
7. Maharjan, R.; Chy, M.S.H.; Arju, M.A.; Cerny, T. Benchmarking Message Queues. *Telecom* **2023**, *17*, 298–312. [CrossRef]
8. The Linux Foundation. OpenMessaging Benchmark Framework. Available online: https://openmessaging.cloud/docs/benchmarks/ (accessed on 17 December 2022).
9. Advanced Message Queueing Protocol (AMQP). Available online: https://www.amqp.org/ (accessed on 12 October 2023).
10. Vinoski, S. Advanced Message Queuing Protocol. *IEEE Internet Comput*. **2006**, *10*, 87–89. [CrossRef]
11. OpenWire. Available online: https://activemq.apache.org/openwire.html (accessed on 13 October 2023).
12. ActiveMQ Artemis Architecture. Available online: https://activemq.apache.org/components/artemis/documentation/1.0.0/architecture.html (accessed on 25 November 2023).
13. Maheshwari, P.; Pang, M. Benchmarking message-oriented middleware: TIB/RV versus SonicMQ. *Concurr. Comput. Pract. Exp.* **2005**, *17*, 1507–1526. [CrossRef]
14. TIBCO Software Inc. TIBCO Rendezvous. Available online: https://www.tibco.com/products/tibco-rendezvous (accessed on 10 October 2023).
15. Progress Software Corporation. SonicMQ Messaging System. Available online: https://docs.progress.com/bundle/openedge-application-and-integration-services-117/page/SonicMQ-Broker.html (accessed on 10 October 2023).
16. Kounev, S.K. SPECjms2007 Benchmark Framework. Available online: https://www.spec.org/jms2007/ (accessed on 10 October 2023).
17. Ahuja, A.; Jain, V.; Saini, D. Characterization and Benchmarking of Message-Oriented Middleware. *Real-Time Intell. Heterog. Netw. Appl. Chall. Scenar. IoT HetNets.* **2021**, *9*, 129–147.
18. Sachs, K.; Kounev, S.; Bacon, J.; Buchmann, A. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. In *Performance Evaluation*; Elsevier: Amsterdam, The Netherlands, 2009; Volume 66, pp. 410–434. [CrossRef]
19. Pivotal Software. RabbitMQ. Available online: https://www.rabbitmq.com/ (accessed on 12 October 2023).
20. John, V.; Liu, X. A survey of distributed message broker queues. *arXiv* **2017**, arXiv:1704.00411.
21. Vineet, J. Flotilla. Available online: https://github.com/vineetjohn/flotilla (accessed on 25 November 2023).
22. Redis. Available online: https://redis.io (accessed on 20 January 2023).
23. Sarhan, Q.I.; Gawdan, I.S. Java Message Service Based Performance Comparison of Apache ActiveMQ and Apache Apollo Brokers. *Sci. J. Univ. Zakho* **2017**, *5*, 307–312. [CrossRef]
24. Apache Apollo. Apache Apollo. Available online: https://incubator.apache.org/projects/apollo.html (accessed on 12 October 2023).
25. Dobbelaere, P.; Kyumars, S.E. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, Barcelona, Spain, 19–23 June 2017; pp. 227–238. [CrossRef]
26. Sanika, R. Performance Comparison of Message Queue Methods. Master's Thesis, University of Nevada, Las Vegas, NV, USA, 2019. [CrossRef]
27. Milosavljevic, M.; Matic, M.; Jovic, N.; Antic, M. Comparison of Message Queue Technologies for Highly Available Microservices in IoT. Available online: https://www.etran.rs/2021/zbornik/Papers/105_RTI_2.6.pdf (accessed on 25 November 2023).
28. Souza, R.D.A. Performance Analysis between Apache Kafka and RabbitMQ. Available online: http://dspace.sti.ufcg.edu.br:8080/jspui/bitstream/riufcg/20339/1/RONAN%20DE%20ARAU%CC%81JO%20SOUZA%20-%20TCC%20CIE%CC%82NCIA%20DA%20COMPUTAC%CC%A7A%CC%83O%202020.pdf (accessed on 25 November 2023).

29. Klein, A.; Stefanescu, M.; Saied, A.; Swakhoven, K. An experimental comparison of ActiveMQ and OpenMQ brokers in asynchronous cloud environment. In Proceedings of the 2015 Fifth International Conference on Digital Information Processing and Communications (ICDIPC), Sierre, Switzerland, 7–9 October 2015; pp. 24–30. [CrossRef]
30. Prometheus. "Prometheus Documentation". Available online: https://prometheus.io/docs/introduction/overview/ (accessed on 19 November 2023).
31. Labs, Grafana. "Grafana Documentation". 2018. Available online: https://grafana.com/docs/ (accessed on 19 November 2023).
32. RocketMQ Discussion Thread. Available online: https://github.com/apache/rocketmq/discussions/7015#discussioncomment-6425354 (accessed on 25 November 2023).