A Comparative Study of Kafka and NATS

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
June 2025
Nino Salonen

Supervisors:
Antero Järvi
Jarko Papalitsas

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Nino Salonen: A Comparative Study of Kafka and NATS

Master of Science (Tech) Thesis, 52 p., 1 app. p.
Software Engineering
June 2025

Distributed messaging systems are a key component of modern software architectures. This thesis compares two such systems to identify the differences and tradeoffs between them. These results aim to help practitioners select the most suitable messaging system for their specific needs.
Kafka is widely used in data-intensive applications for high throughput and persistent messaging, while NATS has traditionally focused more on lightweight real-time communication. With the addition of the JetStream persistence layer, NATS now has capabilities for message durability and replay, making it a viable alternative to Kafka.
This thesis consists of a literature review and an empirical benchmark. The literature review explores the architectural differences and feature sets of both systems for event streaming. The benchmark measures latency, throughput, and resource usage while sending messages in both a single-node and a three-node environment with increased persistence.
The results show that Kafka's design and features for high throughput enable it to achieve overall better performance while also using resources more efficiently. This advantage is more evident under heavy distributed workloads. NATS requires significantly less configuration effort and offers greater flexibility, while also performing reasonably well compared to Kafka. The findings indicate that Kafka's main strength is handling large volumes of messages, while NATS is a viable and more flexible alternative when maximum throughput is not the primary goal.

Keywords: messaging middleware, Kafka, NATS, event streaming, distributed systems

Contents

1 Introduction 1

2 Overview of Distributed Messaging Systems 4

2.1 Advantages in Multi-Application Architectures 4

2.2 Core Components and Features 5

2.3 Monitoring 10

2.4 Common Messaging Patterns 11

3 Survey of Kafka and NATS 12

3.1 Core Features of Kafka 12

3.2 Core Features of NATS 18

3.3 Qualitative Comparison of Kafka and NATS 26

3.3.1 Advantages of Kafka 28

3.3.2 Advantages of NATS 28

4 Performance Metrics of Kafka and NATS 30

4.1 Principles for Benchmark Design 30

4.2 Measured Metrics 31

4.3 Environment Setup 33

4.4 Configuration for Kafka and NATS 34

4.5 Benchmark Results 36

4.5.1 Storage 36
4.5.2 Disk Activity 37
4.5.3 CPU Usage 40
4.5.4 Memory Usage 42
4.5.5 Latency 44
4.5.6 Throughput 45
4.5.7 Warmup Effects 46
4.5.8 Impacts of Push Consumers and Async Publishing 47
4.6 Evaluation of the Benchmark 48

5 Conclusions **50**

References **53**

Appendices

A Use of Generative AI **A-1**

1 Introduction

In modern software architecture, particularly in microservice-based systems, loosely coupled asynchronous communication methods are crucial for facilitating interactions between independently deployed services. Asynchronous messaging patterns enable services to communicate decoupled, allowing applications to operate in their own environments without concerns about compatibility issues or growing integration costs. Message-Oriented Middleware (MOM) is the type of technology that handles this essential communication layer. [1] This thesis focuses on distributed messaging systems, which represent a subset of MOM.

There are many viable solutions for distributed messaging, with Kafka being one of the most widely adopted for event streaming. Event streaming refers to the continuous capture, processing, and storage of event data as it occurs in real-time [2]. Historically, NATS has not been considered a direct competitor to Kafka due to its lack of persistence and a focus on lightweight messaging and real-time communication. However, in March 2021 a persistence layer called JetStream was officially released for NATS [3]. As of today, NATS offers capabilities that allow it to compete more directly with Kafka in event streaming, with features such as persistence and message replay.

This thesis is primarily motivated by the potential of NATS to serve as a lightweight and easily configurable persistent messaging solution for microservice applications instead of Kafka. In addition to messaging, NATS offers features such

as service discovery, key-value storage, and support for lightweight communication patterns. These capabilities could reduce the need for multiple separate technologies, enabling a more streamlined software architecture.

From an academic perspective, another point of motivation for this thesis is the clearly limited literature available on NATS. This highly contrasts to the extensive research on other messaging systems, particularly Kafka. This thesis aims to address this gap and contribute to academic literature by exploring the differences, benefits, and drawbacks of NATS as a persistent messaging middleware. It does not focus on the additional features of NATS.

The thesis consists of a literature review and an empirical benchmark. The literature review is presented in chapters 2 and 3. Chapter 2 provides an overview of distributed messaging systems and their common components and features. Chapter 3 explores the architecture and features of Kafka and NATS, comparing the differences between them. This literature review provides the foundation for understanding the concepts behind Kafka and NATS, as well as their theoretical capabilities and differences.

In Chapter 4, key performance metrics of Kafka and NATS related to event streaming capabilities are measured in a controlled test environment. The metrics include throughput, latency, and resource utilization, such as CPU, memory, and disk usage. The benchmarks are designed to simulate two different server-side configurations. The goal is to provide a comparative evaluation of both systems under similar conditions, rather than to determine their absolute performance limits.

The main objective of this thesis can be summarized into the following research questions:

- RQ1: How suitable is NATS for event streaming based on its features, and how does it compare with Kafka?
- RQ2: How does NATS compare to Kafka in event streaming in terms of

performance and resource footprint?

The first research question is explored in Chapter 3, while the second is explored in Chapter 4.

2 Overview of Distributed Messaging Systems

2.1 Advantages in Multi-Application Architectures

Connection-oriented communication is an efficient method for transferring data between entities. It can involve opening a socket between two entities and sending data through this connection [4]. However, modern applications have become more complex due to diverse business requirements, which are addressed by multi-application architectures or microservice architectures [5]. In such applications, connection-oriented communication may not be sufficient, as it relies on the following assumptions [4]:

1. **Temporal dependency**: All application components must be available at the same time.
2. **Location**: Each application component must know the address of the other component.
3. **Data structure and representation**: The application components must agree on the data exchange format.

Distributed messaging systems address these challenges by decoupling components in different ways. In this context, coupling can be defined as the amount of

assumptions components make about each other in order to communicate.

Distributed messaging systems enable asynchronous communication, allowing for synchronization decoupling [6], [7]. This means that the component sending the message is not blocked during the creation and transmission of messages, while the receiving components can be notified about new messages while actively performing other tasks. Asynchronous communication also allows for time decoupling [6], [7], meaning messages can be sent and received independently of whether the receiving component is available. If the receiver is unavailable, the message can be stored until it is ready to be received. This allows for brief disruptions such as restarts, crashes, or updates, without interrupting the overall system's functionality. In addition, distributed messaging systems eliminate the need for components to know the addresses of other components, a concept known as space decoupling [6], [7]. Messages are routed through addressable channels that are managed by the messaging system, which reduces direct dependencies between components. Finally, standardized message formats eliminate the need for shared data handling logic, ensuring that each component can focus on the contents of communication rather than the details of how it is shared. [4]

2.2 Core Components and Features

In distributed messaging systems, a range of key components and features define how messages are exchanged between clients. This section explores the following: message structure, producers, consumers, communication channels, durability, message ordering, transactions, delivery guarantees, replication, security, consuming models, batching, real-time processing, compression, and scalability.

A message is the basic unit of information exchanged between components and it consists of a header and a body. The message header contains metadata, such as the message ID, size, validity period, destination, and any additional user-defined

attributes. The message body contains the actual content or payload of the message. [5], [6]

Producers and *consumers* are the primary participants or clients that exchange data in the form of messages. Producers create messages, while consumers receive them. When a producer creates a message, the messaging system routes it to the appropriate consumer using a specified communication channel, utilizing transport models such as point-to-point or publish-subscribe. [4], [5]

Communication channels vary across different messaging systems and may be referred to as topics, channels, subjects, and streams [8]. For simplicity, the term "communication channels" will be used to refer to all of these. They serve as intermediaries for message exchange and offer a range of features depending on the messaging system, such as persistence or message ordering guarantees [4]. The messages in these communication channels can be stored either in memory or on a persistent medium [5].

Durability or *persistence* refers to the ability of a messaging system to store messages on a persistent medium, such as a file system or database to ensure that messages are long-lasting and not easily lost. This secures message availability even in the event of system or hardware failures. [4], [5], [9] The level of persistence is typically configurable [5].

Message ordering refers to the capability of delivering messages to consumers in the same sequence they were originally produced [4]. Message order is commonly important [5] because the order of processing can affect the outcome of the system. In some messaging systems, ordering is only guaranteed up to a certain point and can be achieved at different levels:

1. **No order**: Consumers receive messages in random order. [10]

2. **Publisher order**: All consumers receive messages independently in the same order they were published by a single producer. [10]

3. **Causal order**: Consumers receive messages in an order that respects causal dependencies, meaning if one message causally precedes another, all consumers will observe them in that order. However, independent messages may be received in different orders by different consumers. [10]

4. **Total order**: All consumers will receive messages in the same order. Total order does not guarantee publisher order or causal order alone. [10]

Transactions ensure that a set of operations, including sending and receiving messages within a messaging system, is treated as a single unit of work. The operations are either fully committed together or not applied at all in case of any failures. [6], [11] Transactions can guarantee that messages are not lost or duplicated, and they are particularly useful in mission-critical applications requiring strong consistency [6]. By integrating transactions with other operations, such as database updates, they provide end-to-end consistency across distributed systems [12], [13]. Transactions can also be used to ensure exactly-once message delivery [12].

Messaging systems provide different *delivery guarantees* to define how reliably messages are transferred from producers to consumers. The three main types are *at-most-once*, *at-least-once*, and *exactly-once* delivery. At-most-once delivery is the most efficient, but offers no guarantee that a message will reach its destination, potentially leading to message loss. At-least-once delivery ensures that messages are received but may result in duplicates if acknowledgments are delayed or not received properly. Exactly-once delivery is the most reliable, ensuring that each message is delivered only once without duplication, though it comes with additional overhead. [10], [14], [15]

Replication in messaging systems is a mechanism to maintain redundant copies of messages or other metadata across multiple brokers or nodes, ensuring that the system is fault tolerant and always available despite failures in single brokers or nodes. [11] Replication can be synchronous or asynchronous. In synchronous repli-

cation, the leader node waits for the follower nodes to acknowledge the reception of messages before confirming success to the sender [5]. This approach offers stronger consistency but introduces higher latency. In asynchronous replication, the leader node confirms success without waiting for acknowledgments from follower nodes [5]. This approach leads to lower latency but increases the risk of inconsistencies. Regardless of the approach, maintaining replicas introduces performance overhead due to the need to copy data.

Messaging systems can offer *security* features depending on their type and use case. Various security guarantees may be provided, such as controlling who can publish data to communication channels, who can subscribe to these channels, and how data within the channels is transferred [1]. Security services can be implemented at different levels. First, at the network level, where system nodes authenticate with each other. Second, at the transport level, where data packets are encrypted and decrypted during transfer. Third, at the application level, where messages themselves may be encrypted and decrypted. [5], [15] Each level may offer a default security implementation, but they can also support custom security configurations [5].

The *consuming model* defines how messages are delivered to consumers and can be either push-based or pull-based. In the push model, messages are actively sent to consumers as soon as they become available on the server [14]. This approach can reduce latency, since consumers do not need to periodically request new messages. However, if not properly balanced, the push model may flood the consumers with too many messages [16]. In the pull model, consumers request new messages only when they are ready to process them. This approach allows each consumer to retrieve messages at the maximum rate they can sustain, while making it easier to rewind messages, as consumers keep track of their position in the message stream. [16]

Messages can be transferred in real-time or in batches [14]. In general, choosing

between them involves a compromise between latency and throughput. *Real-time processing* allows for minimal latency by sending or receiving data as soon as it is available. However, this approach can introduce additional overhead due to frequent network roundtrips. In *batch processing*, messages are collected and processed as chunks. By grouping messages together, batch processing can increase throughput and reduce network transmission overhead. [17]

An additional technique to reduce network congestion and save disk space is message *compression*, which is offered by some messaging systems [5]. Combining compression and batching can significantly decrease network traffic, with compression becoming more effective as batch size increases [18].

Distributed messaging systems need to be able to scale. In this context, *scaling* refers to the capability of the system to handle a growing number of clients [9] or messages. A system can scale horizontally or vertically. In horizontal scaling, the amount of destinations are added to keep the load manageable in a single destination. The messaging system handles the coordination between different destinations. In vertical scaling, a system handles the growing number of messages by allowing existing destinations to utilize more resources or by increasing the physical resources in those destinations. [12] Horizontal scaling is a key feature in distributed messaging systems, and it can be separated into two categories: scaling consumers and scaling server nodes.

Scaling server nodes horizontally, also known as clustering, may become important when the messaging system needs to handle a higher volume of incoming messages, increase storage capacity, or support more clients. By adding more server nodes, the system can distribute the load more efficiently, ensuring higher availability and fault tolerance across the network [5]. Similarly, scaling consumers horizontally becomes important when one or more consumers can't handle the incoming message load. Adding more consumers helps distribute the workload and improves fault tol-

erance, as other consumers can take over if one fails to ensure uninterrupted message processing [14].

2.3 Monitoring

Monitoring helps make decisions based on data, quickly address issues, and optimize resource usage. Effective monitoring consists of the following features: logging, metrics, tracing, and alerts, all of which should be integrated into a same platform for a comprehensive view. Logging provides insights into past events and errors to help diagnose issues and identify anomalies. Metrics provide quantitative data on key indicators about the system, such as resource usage, as well as application-specific data, like queue size, latency, and health status. Tracing provides visibility into how messages flow through services, helping to identify bottlenecks. Alerts notify of critical conditions or events in real time, such as growing queue sizes or increasing latencies. They help prevent issues from escalating and enable quick reactions to critical events. [19]

While monitoring is crucial, not every messaging system implements monitoring features to the same extent. Some offer comprehensive solutions with built-in web interfaces, dashboards, and alerting mechanisms, while others focus primarily on exposing interfaces and data points that allow integration with third-party monitoring services [5].

Having features like alerting and tracing are valuable, but not essential, as well-established third-party solutions and ecosystems exist that can be integrated into current messaging systems to provide these functionalities. [19] From the messaging system's perspective, exposing metrics and having configurable and comprehensive logging is the most essential. When relevant metrics are exposed, they can be scraped and collected using third-party tools. This metric data can then be used for various purposes by other software, such as alerting. In contrast, manually

extracting system and application metrics can be complex, if not impossible. The same applies to logging. When relevant logs are provided at configurable logging levels, they can be aggregated and analyzed using various log aggregators. If the system fails to effectively expose logs, aggregation becomes ineffective and loses value, as third-party software logs cannot be modified or enhanced.

2.4 Common Messaging Patterns

There are several patterns that can facilitate communication between clients. In this section, the point-to-point model, the publish-subscribe model, and the request-response model will be discussed.

The *point-to-point model*, also known as the queue model, involves producers sending messages to a specific queue, where they are consumed by consumers in sequence. In the case of multiple consumers, each message is typically processed by only one consumer. If no consumers are available, messages are retained in the queue until a consumer can retrieve them. [4]–[6]

The *publish-subscribe model* allows publishers to broadcast messages to a communication channel, to which subscribers can subscribe to receive messages. Producers do not need to know the specific consumers, and consumers only need to know the communication channels they are interested in. [5], [15] There are two types of publish-subscribe subscriptions: durable and non-durable. In the durable model, messages are retained even if the subscriber is offline [5]. With non-durable subscriptions, the subscriber has to be online to receive messages [5].

The *request-response model* allows a requester to send a message and wait for a response from a responder [14], [15]. This can be implemented synchronously, where the requester blocks until it receives the response, or asynchronously, where the requester can continue while waiting for the response in a separate thread or process [14].

3 Survey of Kafka and NATS

3.1 Core Features of Kafka

Kafka was released in 2011 by LinkedIn to address the challenges of processing large volumes of log data. The log data included offline analytics, but also operational metrics and real-time user events to power features like recommendations, search, and ad targeting. Existing systems at the time offered unnecessarily complex features for real-time support, such as transactional support and per-message acknowledgments, lacked support for distribution across multiple machines, did not prioritize high throughput enough, and lacked support for producers to batch messages. Kafka aimed to address these issues by offering lower latency, higher scalability, and high throughput while retaining the benefits of offline-first messaging systems. [16] Kafka is written in Java and Scala, which means it runs on the Java Virtual Machine [20].

Kafka consists of three basic components: producers, brokers, and consumers. Producers publish messages to topics, while consumers subscribe to topics and read messages from them. *Topics* are categories that serve as the communication channels in Kafka. Consumers and producers are fully decoupled and they are not aware of each other, which is why brokers manage the storage and retrieval of messages. Multiple Kafka brokers together form what is known as a Kafka cluster. [16], [21]

In Kafka, a message, generally referred to as a *record*, is fundamentally a payload

of bytes. A message consists of a key, timestamp, value, and optional metadata headers. Messages do not have an explicit ID. Instead, they are addressed by their logical offset within the record log. The offset of the next message can be computed by adding the length of the current message to its offset. [16], [21]

A topic can be split into several *partitions*, and these partitions can be distributed among the brokers in a Kafka cluster. A partition contains an ordered, immutable sequence of records, and it is physically implemented as a set of segment files on the file system. Because partitions are stored on disk, Kafka only offers on-disk storage for messages, not in-memory storage. When a producer sends a message to a topic, the broker appends the message to the most recent segment file. These segment files are typically flushed to disk after a specified number of messages or after a certain time interval has elapsed to optimize for performance. Only after flushing, the messages are available to consumers. Messages are deleted from disk after a configured retention period. [16], [21] Figure 3.1 illustrates an example of a Kafka cluster with two brokers and a single topic containing two partitions.

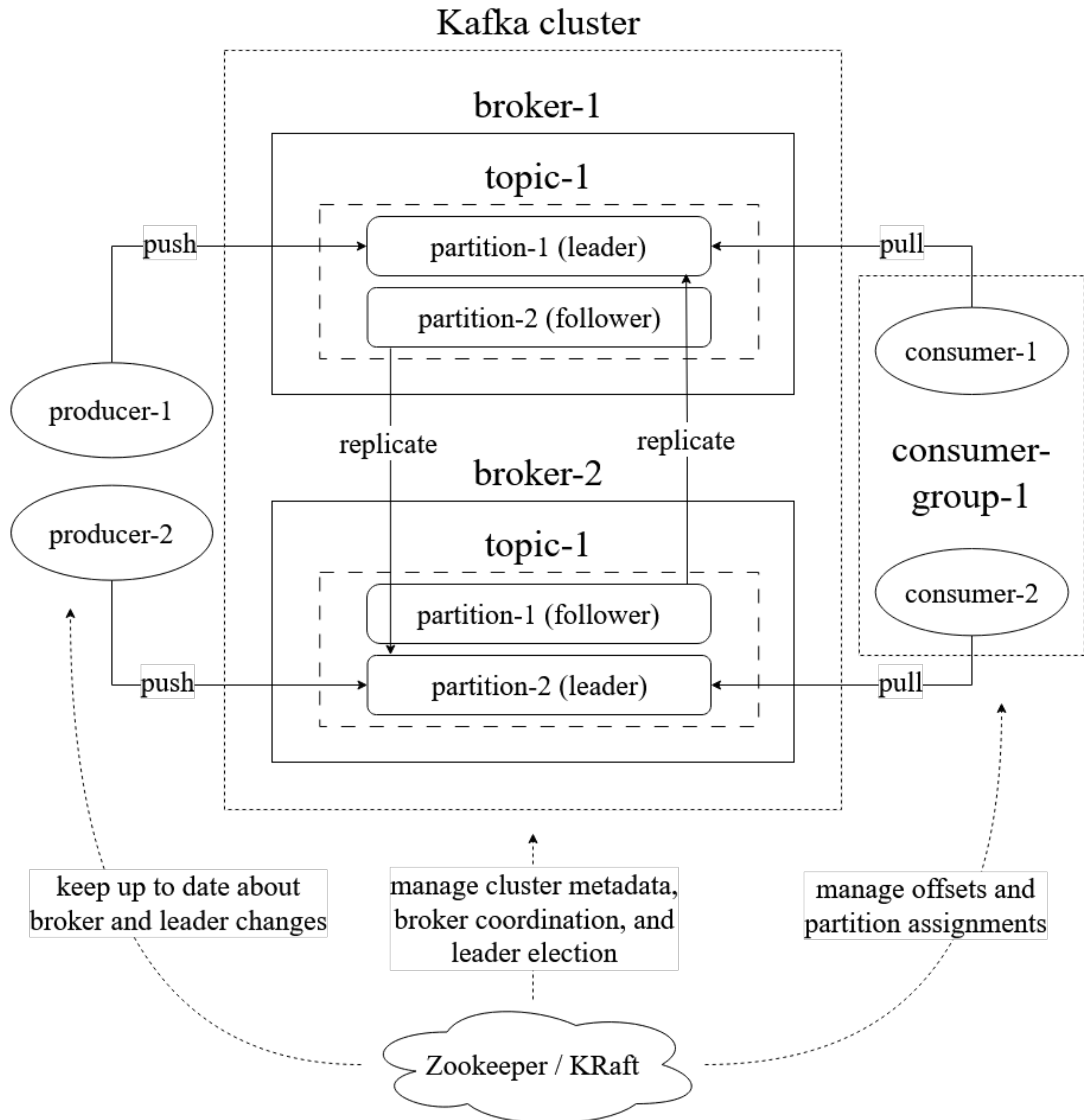

Figure 3.1: Example Kafka Architecture

Kafka provides per-partition total ordering [22] and publisher ordering [21]. When multiple consumers consume from the same partition, each one will receive all messages in the same order they were written to the partition. No ordering guarantee exists for messages originating from different partitions on the same topic. Achieving topic-level message ordering would require partitions to coordinate with each other,

introducing delays and significant overhead. The lack of topic-wide ordering is a tradeoff by design, as partitions are designed to operate independently to achieve higher scalability and throughput. To ensure message ordering, producers can send messages to a specific partition based on the record key, also known as partition key. If no key is present, messages are distributed across partitions randomly. [16] In Figure 3.1, producer `producer-1` uses a specific partition key to send messages to partition `partition-1`, while producer `producer-2` uses a different partition key to send messages to partition `partition-2`.

Consumers process messages sequentially using a pull-based consumption model and track their reading progress by maintaining the offset of the last consumed message. A consumer receives data from the broker by sending a pull request, specifying the starting offset and the amount of data it wants to receive. The consumer manages the current offset, and the Kafka broker is not aware of it, which simplifies rewinding to a previous point in the stream for reprocessing data. This approach allows the consumer to control the amount of data it receives, and by requesting larger chunks at once, it reduces the number of interactions with the broker, lowering the overhead of broker-consumer communication. Producers can also batch multiple messages into a single publish request, further reducing the overhead of producer-broker communication. [16] In addition to efficient batching and high throughput, Kafka is well-suited for real-time processing using the Streams API. Latencies under 100ms can be achieved, making it ideal for applications that require near-instantaneous responses to data [23].

Kafka supports GZIP, Snappy, LZ4, and ZStandard compression. The producer can compress messages before sending them to the broker, which decompresses the messages for validation. The messages are then stored in the event log in their compressed form. Consumers can also receive messages in the compressed form. Compression becomes more efficient as the message batch size increases and is less

effective when sending messages individually. [21] The combination of batching and compression, on both the producer and consumer sides, can significantly improve throughput by reducing network congestion.

Kafka supports all three types of message delivery guarantees, with at-least-once being the default. In at-most-once delivery, producers send messages without waiting for confirmation, and consumers commit offsets before processing messages. In at-least-once delivery, producers resend messages if they don't receive a confirmation. Consumers in this mode process messages before committing their offsets. For exactly-once delivery, Kafka uses transactional producers and consumers that track messages and offsets together. [18]

To facilitate parallel consumption and load balancing, Kafka introduces the concept of consumer groups. A *consumer group* consists of one or more consumers that collectively consume all the messages within the subscribed topics. Each message is delivered to only one consumer within a single consumer group. So to achieve parallel consumption, multiple consumer groups can be created, because different consumer groups operate independently and each group receives its own copy of all subscribed messages. Additionally, consumer groups help distribute load by allowing more consumers to be added to a group. [16] In Figure 3.1, the consumer group `consumer-group-1` is configured to consume messages from topic `topic-1`, ensuring that both consumers in the group are automatically consuming from separate partitions.

With consumer groups, Kafka supports the point-to-point delivery model in addition to the publish-subscribe model, since only a single consumer within a consumer group can read from a partition at a given time. This is why a partition can be considered as the smallest unit of parallelism in Kafka, and it is by design, because allowing multiple consumers to read from a single partition simultaneously would require increased coordination overhead that would lower throughput. It would also

make it more difficult to maintain message ordering at the partition level. [16]

Kafka scales horizontally by adding brokers to the Kafka cluster. Multiple brokers allow partitions to be replicated across several brokers. When a topic is created, a replication factor specifies how many copies of each partition will be maintained across different brokers. For each partition, one broker is designated as the leader, handling all read and write requests, while follower brokers fetch data from the leader to maintain an identical copy of the partition logs. When the leader receives a new record, it is considered successfully committed when all in-sync replicas (ISR) have applied the record to their log. Consumers cannot receive messages that are not successfully committed. If the leader broker fails, Zookeeper (in older Kafka versions) or KRaft (in newer Kafka versions) automatically elects a new leader from the follower brokers. [16], [18], [21] In Figure 3.1, each partition has a replication factor of two, with broker `broker-1` acting as the leader of partition `partition-1`, and broker `broker-2` acting as the leader of partition `partition-2`.

Zookeeper is a consensus service that Kafka has historically used to coordinate brokers. Instead of using a central master node to coordinate everything, the approach with Zookeeper eliminates the potential for a single point of failure in master nodes. Zookeeper is a separate process that monitors changes in the cluster. It detects the addition or removal of brokers, ensuring the state of the cluster remains up to date. [16] As of Kafka 4.0, Zookeeper has been replaced by *KRaft* to improve scalability and simplify Kafka's architecture. Before, Kafka needed both Zookeeper and Kafka to manage metadata, which made deployment and management more complicated. With Zookeeper, Kafka clusters were limited to about 200000 partitions due to the time it took to move critical metadata between Zookeeper and Kafka's internal leader management, but KRaft can handle up to two million partitions with significantly better performance in terms of shutdown and recovery times. [18], [21]

Kafka provides multiple security features to control access, protect data in tran-

sit, and authenticate clients and brokers. Mutual authentication between brokers and clients is supported using SSL/TLS or SASL. Additionally, SSL/TLS encryption can be enabled to secure data exchanged between brokers, clients, and other components. At the application level, Kafka offers authorization mechanisms using ACLs (Access Control Lists) to regulate read and write permissions. All of these security features are optional, allowing the use of clients with various security levels, ranging from full authentication and encryption to no authentication or encryption. [21]

By default, Kafka uses the Log4j framework for logging [21], allowing control over log levels, output destinations, and formats. Metrics are exposed via Java Management Extensions (JMX), which is a standardized Java framework for exposing application metrics. JMX provides access to essential system information and also Kafka-specific metrics. JMX is highly extendable, allowing it to integrate with various monitoring and management tools, such as JMX exporter, which enables Prometheus to scrape JMX metrics [24], [25]. Prometheus is a widely used open-source monitoring solution and it can be used to collect and store metrics [26].

3.2 Core Features of NATS

NATS was initially released in 2011, but in 2012, the original Ruby codebase was rewritten in the Go programming language to enhance performance. The primary motivation behind NATS was to overcome the limitations of point-to-point inter-service communication within a platform called Cloud Foundry. In large multi-application architectures, direct point-to-point connections are inefficient, as discussed in Section 2.1. To address this, NATS manages communication by decoupling clients and offering high performance through a simple interface, which quickly gained popularity within the developer community. [14]

In NATS, publishers send messages to specific subjects, while subscribers sub-

scribe to these subjects to receive messages. Subjects act as the communication channels in NATS, and the server acts as a central connection point, routing messages from publishers to subscribers based on the subject. [14] Multiple servers can be connected in a mesh topology to form a cluster, and multiple clusters can be connected to form a supercluster [8].

A *subject* is a named category that the server uses to forward messages from publishers to subscribers in a one-to-many communication pattern. NATS uses interest-based messaging, delivering messages only to clients subscribed to the specific subject. [14] Subjects are ephemeral and persist only while clients are subscribed to them [8]. Figure 3.2 illustrates an example, where the subscribers `subscriber-1` and `subscriber-2` receive both messages published to subject `subject.one`.

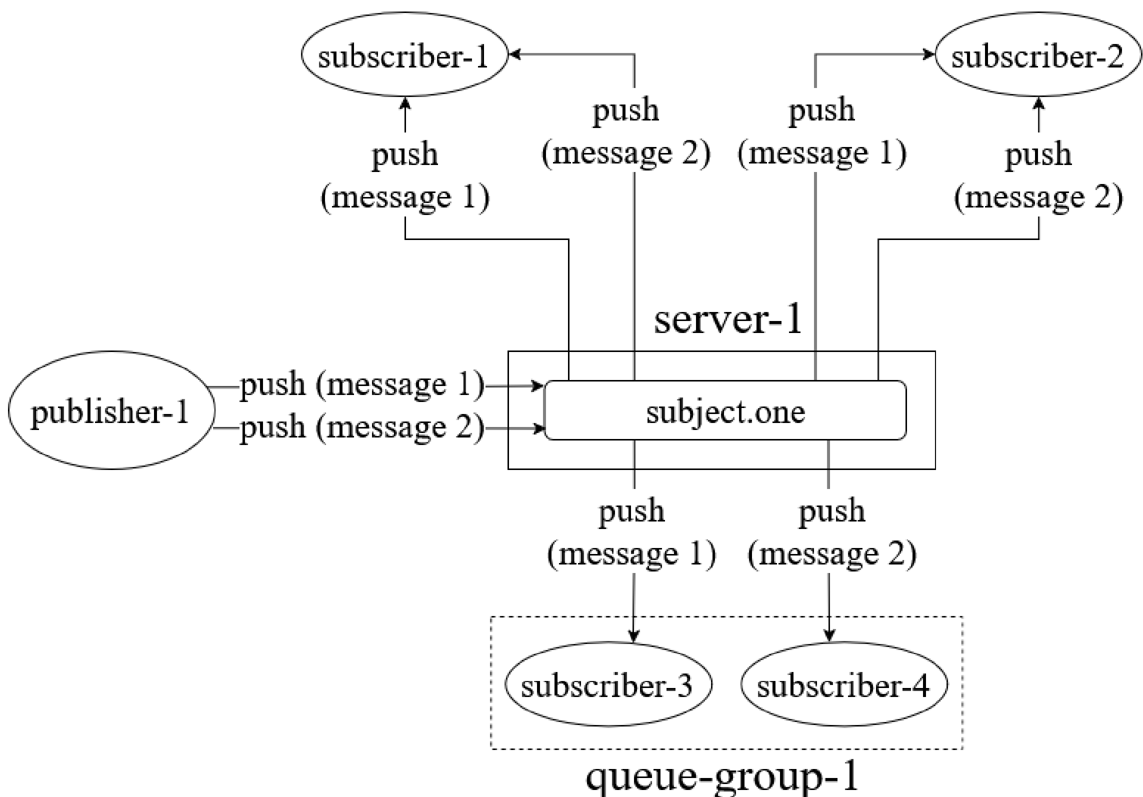

Figure 3.2: Example NATS Architecture without Persistence

A subject is simply defined as a string of characters. Subjects can form hierarchies using the dot "." character to separate different levels, creating a logical structure. For example, `orders.eu.delivered` and `orders.us.cancelled` are two different subjects within the `orders` hierarchy. Subscribers can use wildcards like `orders.*.cancelled` to receive messages about all cancelled orders around the world, or `orders.>` to receive messages about all order statuses globally. [8], [14]

Queue groups in NATS enable load balancing and point-to-point communication by distributing messages from a subject among multiple subscribers. When subscribers register under the same queue name for a subject, only one subscriber from the group will receive each message. The number of subscribers in the queue group can be increased to balance the workload more effectively between subscribers. Queue groups won't guarantee message order, as messages are distributed across multiple subscribers, which can process the messages at different speeds. [8] In Figure 3.2, subscribers `subscriber-3` and `subscriber-4` are part of the queue group `queue-group-1`, so only one of the subscribers will receive each message that is published to subject `subject.one`.

A *message* in NATS consists of a destination subject, a byte array for the payload, header fields, and an optional reply subject, also known as the inbox. The payload is in plaintext, and NATS does not enforce any specific format for it. [14] By using the inbox, NATS leverages its publish-subscribe mechanism to support the request-reply pattern. When a request is published to a subject along with an inbox, the reply is dynamically routed back to the requester through the inbox. The reply can then be awaited either synchronously or asynchronously. [8]

When using subjects, publishers, and subscribers alone, the messages are not persisted and there are no guarantees for delivery. NATS allows messages to be stored and replayed using streams and consumers, which are part of the JetStream persistence layer added to NATS. JetStream can be easily enabled with a flag when

message persistence is required. Streams capture and persist messages published to configured subjects, while consumers provide subscribers access to these stored messages. Subscribers can either subscribe directly to a subject or to a consumer. Subscribing to a consumer allows messages to be received in a managed way, with message state and acknowledgements being tracked by the consumer. When the server has JetStream enabled, publishers can receive acknowledgments for message delivery, ensuring higher quality of service and allowing them to retry delivery if needed. [8]

Streams act as message stores that capture and persist messages published to one or more specified subjects, keeping related data together. Streams can use either file-based or memory-based storage and support configurable retention limits based on factors such as duration, stream size, or consumer interest. NATS allows compressing streams using the S2 compression format. However, compression is only available on server-side when storing streams on disk. NATS does not offer a built-in solution for compressing messages transmitted between clients and the server. [8]

Consumers are a view of a stream that track the delivery and acknowledgments of messages to subscribers. They can be either ephemeral or durable. Ephemeral consumers don't persist state. They can only have a single subscriber, and they exist for the lifetime of that subscription. Once the subscriber is no longer connected to receive messages, the consumer is deleted, which can be useful for short-lived sessions. [27] In contrast, durable consumers persist their state on the server, allowing them to recover from failures and resume message processing even after server restarts. Both consumers can also be either pull or push-based. Pull consumers give subscribers control over processing, allow messages to be requested in batches, and automatically scale horizontally with multiple subscribers. In contrast, push consumers actively deliver messages to subscribers as soon as they arrive on the server and do not support batch delivery. A push consumer is created when the

`DeliverSubject` option is set for the consumer. Essentially, clients are subscribing to the deliver subject and not the consumer to receive messages from the consumer. [8]

Figure 3.3 illustrates an example that leverages NATS's persistence features. The stream `stream-1` is configured to capture both subject `subject.one` and subject `subject.two`. The pull consumer `pull-consumer-1` is set to consume the stream, and both subscribers `subscriber-1` and `subscriber-2` are subscribed to the consumer. The pull consumer automatically ensures that the subscribers will process different messages from the stream.

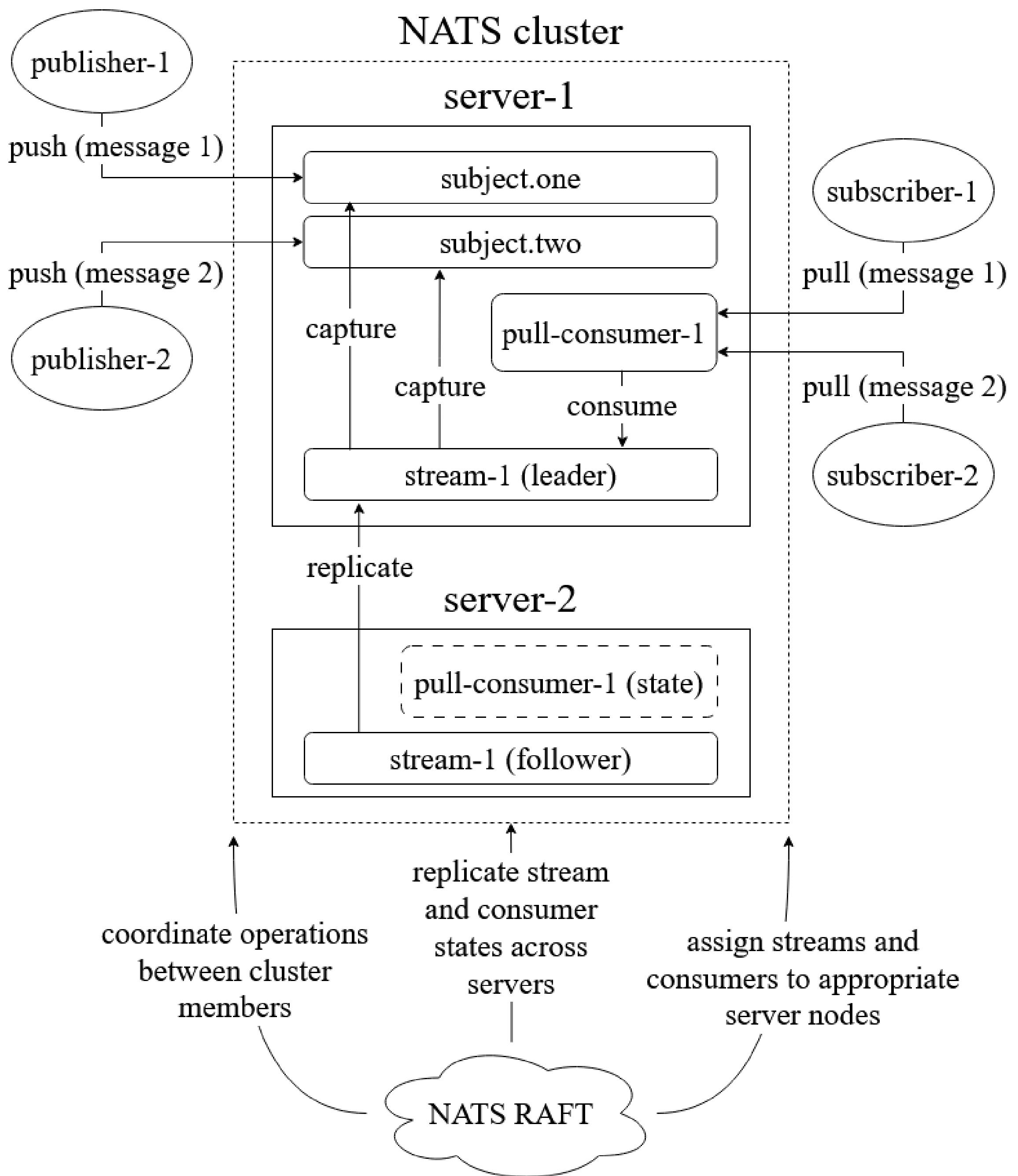

Figure 3.3: Example NATS Architecture with Persistence

Streams and consumers can function as queues for point-to-point communication, offering redelivery and persistence features that queue groups and subjects alone cannot provide. There are two ways to implement this. The first and recommended

approach, as shown in Figure 3.3, is to use a pull consumer with multiple subscribers, where messages are automatically distributed among the subscribers. The second approach involves using a queue group of subscribers and a push consumer with the acknowledgment policy set to explicit. The push consumer automatically sends new messages to the specified deliver subject (`DeliverSubject`). The messages are sent only to subscribers that are part of the defined queue group, specified in the `DeliverGroup` option. Messages are distributed evenly among the subscribers, and any unacknowledged messages are redelivered. In both approaches, the stream's retention policy is set to `WorkQueuePolicy`, ensuring that acknowledged messages are deleted from the stream. [8]

NATS offers all three levels of delivery guarantees. When using subjects alone, messages are published with an at-most-once guarantee, meaning that if subscribers are unavailable, messages may be lost. For at-least-once delivery, a JetStream-enabled NATS server using streams and consumers can acknowledge both publish calls and client deliveries. Finally, exactly-once delivery is supported within a specified time window, which also requires a JetStream-enabled server with streams and consumers. By attaching a message ID header when publishing messages, the server can ignore duplicates within that window. Additionally, double acknowledgments are used, where the server acknowledges the subscriber's acknowledgment of message processing to prevent re-sending. [8] However, a large time window for deduplication is not recommended [8], most likely due to potential performance degradation.

Subjects in NATS have publisher ordering. Messages published to a single subject by a single publisher will be delivered to each subscriber in the same order they were published. For streams, there are no ordering guarantees by default. Ordered consumers can be used with streams to ensure ordered message delivery for a single client instance. These consumers are ephemeral, meaning they do not persist their state and are deleted when the client disconnects. Because they are ephemeral,

ordered consumers cannot recover from client failures. To maintain order, they operate in a single-threaded manner, making them significantly slower. Additionally, they cannot distribute workloads across multiple subscribers since they are tied to a single subscriber during their lifetime. Finally, ordered consumers do not support message acknowledgments, meaning reliable delivery requires recreating the consumer if any gaps are detected in the messages. While it is possible to have multiple clients process messages in the exact same order by creating an ordered consumer for each client, this approach comes with the trade-offs mentioned above. [8]

NATS clustering works by connecting multiple NATS servers, allowing them to automatically discover and connect to each other without requiring extensive configuration. The servers automatically gossip, or share information about the other servers they know, ensuring that all servers in the cluster stay updated. This enables the cluster to grow, shrink, and recover on its own. Clients are also automatically notified of any changes in the cluster, allowing them to connect to other available server nodes if one becomes unavailable. By utilizing a cluster, NATS allows streams to be replicated using the NATS-optimized RAFT consensus algorithm. When a message is published to a stream, it is first written to the leader stream. It is then synchronously replicated to the configured amount of follower streams and considered successfully saved only after a quorum of nodes, specified by the RAFT protocol, has acknowledged the message [28]. [8]

NATS's RAFT protocol manages replication and consensus across multiple servers. It is responsible for managing which servers handle which streams and consumers. All server nodes join a meta group, which is used to manage API calls and cluster-wide operations. Each stream creates a stream RAFT group that is used to replicate its state across cluster members. Similarly, consumers create consumer RAFT groups to synchronize their state across the cluster. [8] In Figure 3.3, server `server-1` is the leader of stream `stream-1` that has a replication factor of two, and server

`server-2` is the follower, meaning server `server-2` replicates the data from the stream and maintains the latest state of consumer `pull-consumer-1`.

NATS provides security measures across multiple levels. It supports mutual TLS authentication and various other authentication methods, including tokens, username and password, NKEY, and decentralized JWT authentication. TLS encryption secures communication between clients and servers. At the application level, NATS enforces access control using subject-level permissions for each authenticated user. With multi-user authentication, permissions define which subjects a user can publish to and subscribe to. A user in this context is a client. NATS also allows integration with custom authentication and authorization systems. [8]

NATS supports configuring various logging levels, redirecting log output, log rotation, and more. For metrics, NATS exposes multiple HTTP endpoints providing information on server status, connections, streams, health status, and more. These endpoints return JSON objects, requiring additional processing for some third-party monitoring tools. [8] However, NATS has an official Prometheus exporter that aggregates data from all HTTP endpoints into a single Prometheus-compatible endpoint [29].

3.3 Qualitative Comparison of Kafka and NATS

All features covered earlier are summarized in Table 3.1. This section highlights the differences that could potentially have an impact, excluding minor differences in how some common functionality is implemented. It is worth mentioning that both systems implement exactly-once delivery in different ways. Kafka uses transactions, while NATS uses deduplication and double acknowledgements. As discussed in Section 2.2, transactions can offer strong consistency guarantees, particularly in distributed environments.

Table 3.1: Comparison Between Kafka and NATS

Property	**Kafka**	**NATS**
Language	Java and Scala	Go
Message routing	Topics with fixed partitions	Subject-based addressing with wildcard support
Messaging models	Publish-subscribe and Point-to-Point	Publish-subscribe, Point-to-Point, and Request-Reply
Consumption models	Pull	Pull and push
Message compression	End-to-end compression with GZIP, Snappy, LZ4, and ZStandard	Server-side compression with S2
Batching support	Consumers and producers	Consumers
Real-time streaming	Yes	Yes
Delivery guarantees	At-most-once, at-least-once, and exactly-once (using transactions)	At-most-once, at-least-once, and exactly-once (using deduplication and double acks)
Persistence	On-disk	On-disk and in-memory
Message order	Total and publisher ordering in partitions	Publisher ordering in subjects
Horizontal server scaling	By adding brokers to a cluster	By adding servers to a cluster
Horizontal consumer scaling	By adding consumers to a consumer group	By increasing subscribers on pull consumers, or by adding subscribers to queue groups that receive messages from push consumers or publishers
Replication	Partitions replicated across brokers	Streams replicated across cluster nodes
Authentication	SSL/TLS and SASL	TLS, Token-based, Username/Password, NKEY, and JWT authentication
Encryption	SSL/TLS	TLS
Authorization	ACLs	Per-user and multi-user permissions
Logs	Log4j	Configurable with built-in options
Metrics	Exposed via JMX	Exposed via multiple HTTP endpoints

3.3.1 Advantages of Kafka

Kafka can batch messages when sending them from producers, whereas NATS does not currently support this feature. However, multiple feature requests for message batching from publishers have been made for NATS, so it may become available in the near future [3]. Batching messages can significantly reduce network traffic and increase throughput.

When batching is combined with compression, larger batch sizes can achieve even greater compression savings. Kafka supports compressing batched messages between clients and the server, while NATS only allows compression when saving messages on the server's disk. Kafka also offers a wider range of compression algorithms, allowing for finer optimizations.

Finally, Kafka guarantees both total ordering and publisher ordering within partitions, while NATS guarantees only publisher ordering within subjects. With total ordering, multiple consumers for a partition receive all events in the exact same order. This ordering is crucial for applications where the sequence of events impacts processing logic. Kafka's approach enables high throughput while also maintaining these ordering guarantees.

3.3.2 Advantages of NATS

NATS's subjects are more flexible compared to Kafka's topics and partitions. Subjects allow creating hierarchies and also support wildcard matching, which allows for more flexibility when sending messages. Subjects are ephemeral, and they can be created on the fly, unlike partitions, which have to be manually defined and configured.

NATS natively supports the request-reply communication model, whereas Kafka does not. This provides flexibility in scenarios where immediate action is required after a message is successfully received and processed. It also allows the reply value

for the message to be used for further processing, such as notifying other clients about the recent change. Since NATS supports asynchronous request-reply, this operation doesn't need to block the main execution while waiting for the response.

NATS also supports in-memory storage for streams, while Kafka only supports storage on disk. This gives NATS flexibility for applications where durability can be sacrificed for performance. In-memory streams operate with lower latencies [30], which is ideal for temporary workloads that require rapid processing. Combining in-memory streams with NATS's ephemeral consumers can be a useful lightweight approach for messaging, where persistence beyond server restarts or crashes isn't required.

Finally, both push and pull consumers can be utilized in NATS, while Kafka only supports pull consumers. Push consumers are beneficial when the latency between the publication and the processing of the message has to be minimized, because separate polling is not needed. If batching is not used and events are usually sent individually, push consumers can help reduce network bandwidth since clients don't have to request new messages. When using push consumers, it's important to ensure that the message flow doesn't exceed the clients' ability to process the messages.

4 Performance Metrics of Kafka and NATS

This chapter details the metrics collected, the server environments and configurations used, and how the clients were executed during testing. The full source code is available on GitHub, including all relevant files and instructions for running the tests [31].

4.1 Principles for Benchmark Design

Designing an effective benchmark for messaging systems requires following a set of principles to ensure the results are meaningful, fair, and reflective of real-world use. At the core of this is a clear motivation or purpose, which ties directly to all other aspects of the benchmark by guiding design choices, goals, and conditions. A good benchmark should be relevant, reproducible, fair, and verifiable, with a transparent process and well-defined performance metrics. There is also the dilemma between creating a benchmark that is highly specialized and meaningful for a narrow domain versus one that is general and widely applicable but less insightful for any specific scenarios. [32]

A *relevant* benchmark aligns closely with real-world scenarios, producing results that are meaningful to the intended audience. This requires a clear understanding of how the software is meant to be used, so the benchmark can simulate those

conditions accurately. Scalability is an important aspect of relevance, as it allows the software to take full advantage of the available resources. [32], [33]

A *fair* benchmark avoids artificial limitations or constraints that could skew the results or favor some software over others. While synthetic benchmarks often require some limitations, these limitations should not create unrealistic conditions. On the other hand, too few restrictions can also lead to unrealistic results. A fair benchmark ensures an even playing field for all evaluated software close to their natural execution environment. [12], [32]

A *reproducible* benchmark is more credible. This requires describing the environment, including hardware specs, system configurations, and software versions. The results should be consistent on different runs, but also similar on other systems using the same specifications. To support this, the benchmark should offer a practical path to replication by clearly specifying the setup and workloads, or by including scripts or tools to recreate the environment. Ideally, it should do both. [32], [33]

A *verifiable* benchmark provides enough information that the results are accurate. This requires clear performance measurements that not only describe the outcomes of the benchmark but also help verify that the benchmark ran as intended. It's better to include more metrics than too few, as this provides a better picture of what happened during the benchmark. [32], [33]

4.2 Measured Metrics

Throughput and latency can be considered one of the most important metrics for messaging systems in general [34], [35]. It is also important to measure how the messaging systems react to increased workloads in terms of resource consumption [34], [35]. The metrics that are collected are the following:

- **End-to-end latency:** The time elapsed between when the publisher or pro-

ducer creates the message and when the subscriber or consumer receives it. This includes any delays in the publisher or producer side, storage and replication delays on the server side, and polling or processing delays on the consumer or subscriber side. Since the benchmark runs locally, any network transmission delays are negligible. Latencies are measured with timestamps inserted into message headers at the time of creating the message. The time difference is calculated upon receipt and the difference is recorded using a high dynamic range histogram (HdrHistogram). The results are reported with the median, 95th percentile, and 99th percentile latencies.

- **Throughput:** The number of messages that are produced, processed, and consumed per second. In addition to taking into account processing times on the server, this also takes into account the abilities of the clients to produce and consume messages. The timer is started right before sending the first message and stopped once all messages have been processed.
- **CPU load:** The percentage of CPU resources used by the messaging system during runtime.
- **Memory usage:** The amount of memory used by the messaging system during runtime.
- **Disk read/write activity:** The volume of data read from and written to disk by the messaging system during runtime.
- **Disk space:** The amount of storage occupied by the messaging system. This includes the installation itself, such as the binaries, configuration files, and all related data like message payloads, metadata, temporary files, and replication data.

CPU load, memory usage, and disk read/write activity are measured from each

container using the `docker stats` command. These metrics are polled at frequent intervals during the benchmark. Disk space is measured once before and once after running the benchmark from each container.

4.3 Environment Setup

Docker was chosen to run the environments due to its ability to create consistent and reproducible setups. Even though the goal of this benchmark is not to measure the maximum performance of both messaging systems, but to compare their performance under similar conditions, there are some performance considerations with containerized environments.

First, there can be significant overhead with write-heavy operations due to how Docker's storage drivers work. This may impact the performance of Kafka and NATS due to their write-heavy nature. However, this can be mitigated by using volumes to bypass the use of storage drivers. Volumes allow containers to write directly to the host's filesystem to persist data. [36] Second, there is some overhead introduced by Docker's use of an additional link layer device in the bridge network mode to connect containers in an isolated way. The amount of overhead is relatively small, and both environments will be impacted by it equally, so this factor will not be addressed. [37]

Docker Compose is a tool that allows managing multiple container applications [38]. It is used to set up the server environments in two different ways: a single-node setup and a three-node setup. In the three-node setup, the nodes are deployed in separate containers, but run within the same local bridge network. No resource limits are set for the containers, allowing the systems to allocate the necessary resources. Each container is assigned its own volume, which is removed after each test run. The system specifications of the host machine used for the benchmarks are listed in Table 4.1.

Table 4.1: Host Machine Specifications

Operating System	Ubuntu 24.04.2 LTS
CPU	Intel Core i7-10850H
RAM	32 GB DDR4 @ 2933 MT/s
Storage	PC611 NVMe SK Hynix 1TB

Client-side traffic is generated by creating multiple clients, each running in separate virtual Java threads. Virtual threads allow NATS clients to publish messages in a non-blocking way, which is important due to the lack of batching support for publishers. Java was chosen to run the clients due to its good support for both client libraries and its portability across environments. The amount of clients and communication channels is specified in Table 4.2. In Kafka, all consumers reading from a single topic are part of the same consumer group. In NATS, each stream has a single one pull consumer, and multiple subscribers pull messages from the consumer.

Table 4.2: Benchmark Setup

Configuration	**Kafka**	**NATS**
Streams/Topics	10 topics	10 streams
Partitions/Subjects	100 partitions (10 per topic)	100 subjects (10 per stream)
Consumers/Subscribers	10 consumers per topic	10 subscribers per consumer

4.4 Configuration for Kafka and NATS

The key difference between the single-node and three-node setups is the number of messages sent and the use of a replication factor of three in the three-node configuration. In the single-node setup, each subject or partition receives 50 thousand messages, which totals five million. In the three-node setup, this is increased to 150

thousand messages per subject or partition, which totals to 15 million messages. At the end of each test run, the number of processed messages is verified using the count from the HdrHistogram. In all cases, the message size is set to one kilobyte. No compression is used on either the server or client side.

With the replication factor being three, the NATS's RAFT protocol considers messages successfully saved after a quorum of nodes acknowledge that a message has been saved, which in this case is two. Publishers receive acknowledgements from the leader after this quorum of acknowledgements has been reached. Kafka achieves the same behavior by setting the in-sync replica count to two at the topic level, ensuring a message is considered committed once both the leader and at least one follower have successfully committed the message. Additionally, Kafka producers are configured to expect all acknowledgements, which means the producer receives acknowledgements from the leader when all in-sync replicas have successfully committed the message.

Both systems operate with at-least-once delivery semantics. Kafka commits offsets asynchronously after processing a batch of messages, and NATS acknowledges messages asynchronously after each message. NATS does offer the possibility to acknowledge a set of messages once by acknowledging only the last message in the batch. However, this approach cannot be used, as it would also acknowledge any earlier messages for other subscribers as well. Because exactly-once delivery semantics are not required, deduplication is disabled for NATS streams by setting a zero-length deduplication window. Also, no in-memory storage is used for NATS, so all scenarios rely on on-disk storage.

Both Kafka and NATS are configured to pull messages in batches up to 500 with a timeout of one second. To allow all NATS subscribers to pull large batches at the same time, consumers have been configured with an increased amount of maximum acknowledgements pending. Otherwise, each subscriber would have to wait for other

subscribers to finish processing their messages before being able to pull more, which would create a significant throughput bottleneck.

Finally, Kafka producers use the default values for batch size and linger, which are 16 kilobytes and zero milliseconds. These settings tell producers to send messages either when the batch size is reached or to send messages immediately if there are no more records to batch. Given the constant availability of messages, it's likely that the batch size limit is reached every time, so the linger value has no effect.

4.5 Benchmark Results

Before highlighting the differences in the benchmark results in this section, it is worth mentioning that NATS required significantly less configuration effort to get the server environments running compared to Kafka. This difference showed especially when setting up the communication between cluster members. Additionally, NATS required less setup for communication channels and clients. On the other hand, the extensive configuration options for clients and communication channels in Kafka can also be considered a strength when comprehensive tuning is required.

4.5.1 Storage

A single NATS server installation at 22 MB was clearly smaller than Kafka's 140 MB. The small size of a NATS installation comes from its early emphasis on simplicity and lightweight nature [14]. However, this gap disappeared quickly as messages were stored, with Kafka eventually gaining the advantage in storage efficiency. The single-node results in Table 4.3 show that after the benchmarks, NATS used 4.1% more storage than Kafka. This means that Kafka stored data 3.9% more efficiently. A slightly larger difference can be seen in the three-node results shown in Table 4.4, where NATS used 4.4% more storage on average across all three nodes, meaning

Kafka stored data 4.2% more efficiently.

Table 4.3: Single-Node Disk Usage

Node	**Before (MB)**	**After (GB)**	**Increase (GB)**
nats-server-1	22	5.54	5.52
kafka-broker-1	140	5.44	5.30

Table 4.4: Three-Node Disk Usage

Node	**Before (MB)**	**After (GB)**	**Increase (GB)**
nats-server-1	22	16.59	16.57
nats-server-2	22	16.60	16.58
nats-server-3	22	16.60	16.58
kafka-broker-1	140	16.01	15.87
kafka-broker-2	140	16.02	15.88
kafka-broker-3	140	16.01	15.87

These results show that Kafka was built to handle and store large volumes of messages from the start, while NATS has added persistence features afterwards. The efficiency of Kafka's storage format becomes more apparent as the message volume increases. In the long term, this can lead to significant cost savings in storage, or allow Kafka to retain messages longer than NATS using the same amount of storage. On the other hand, for environments with very limited available storage and shorter retention periods, such as queue-like workloads where messages are deleted after processing, NATS can be a more suitable option.

4.5.2 Disk Activity

The disk read activity between Kafka and NATS was minimal, with no significant differences observed. These results were excluded from the analysis. However, there were differences in disk write activity. In the single-node results shown in Figure 4.1, the overall write activity was very similar, with Kafka writing a total of 2.3%

more data to disk. This indicates that both systems are able to handle disk writes with comparable efficiency in simple deployments.

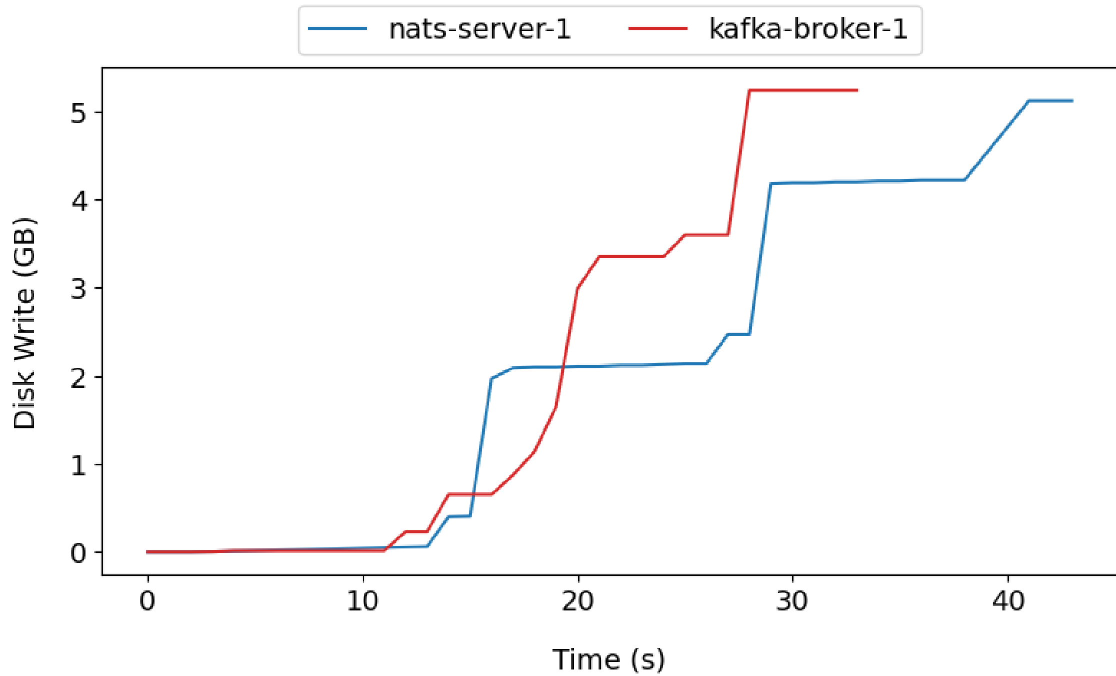

Figure 4.1: Single-Node Disk Write Activity

In the three-node results shown in Figure 4.2, the differences were more significant. Even though the three-node average increase in total storage usage for NATS was 4.4% larger compared to Kafka, the total volume in disk write operations was 22.0% larger compared to Kafka.

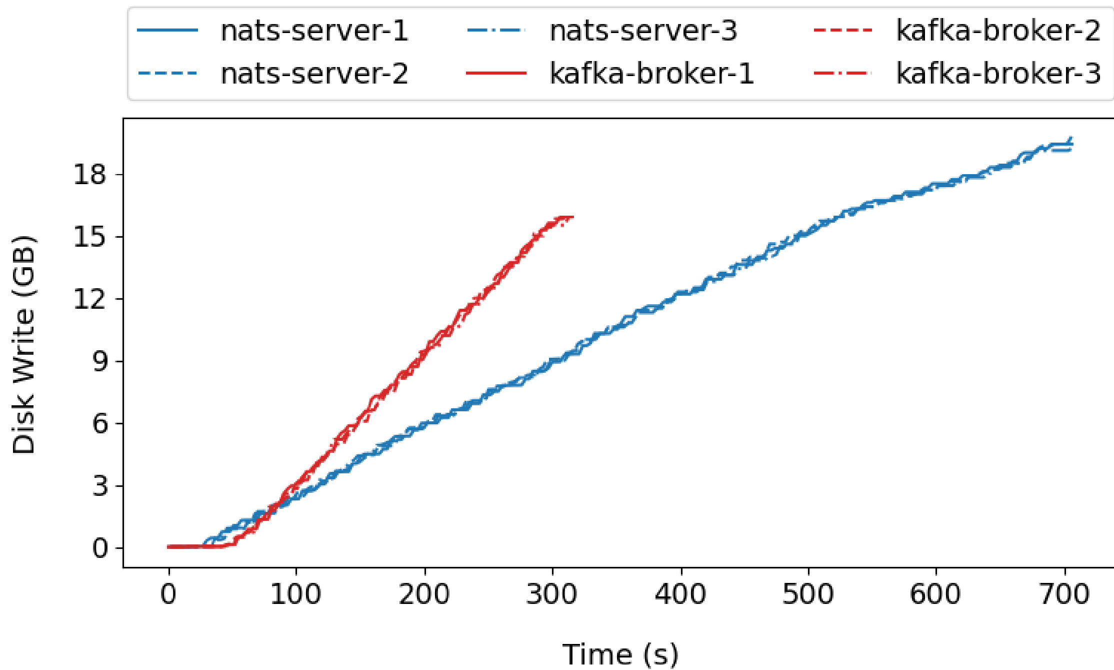

Figure 4.2: Three-Node Disk Write Activity

Because the difference becomes noticeable in the three-node benchmark, it suggests there is more overhead in NATS's RAFT protocol. NATS likely generates more intermediary writes during replication and coordination with other cluster members, while Kafka handles this more efficiently using batching. These differences start to show in distributed settings when message volume increases, where NATS performs more write operations per stored byte.

This has some practical implications in production environments. First, cloud providers may charge separately for disk I/O operations depending on the storage type, which may result in higher costs for NATS. An example of this is Amazon EBS (Elastic Block Storage), where users may be charged for provisioned IOPS (Input/Output Operations Per Second) [39]. Second, more frequent disk writes can increase wear on physical storage devices. This can be relevant in on-premise setups, where hardware replacements lead to more higher maintenance costs. It also raises

environmental concerns due to shorter hardware lifespans and increased electronic waste. Finally, NATS may be more affected in scenarios where disk I/O is the bottleneck.

4.5.3 CPU Usage

In the single-node CPU activity shown in Figure 4.3, Kafka utilized more resources in the earlier half when the producers were most active, resulting in a 29.4% higher 95th percentile CPU usage compared to NATS. Throughout the whole test, NATS's CPU usage was more stable, though its median CPU was 4.2% higher compared to Kafka. Both systems show comparable processing efficiency in this single-node setup, with Kafka having slightly higher peak CPU demands and finishing the task earlier.

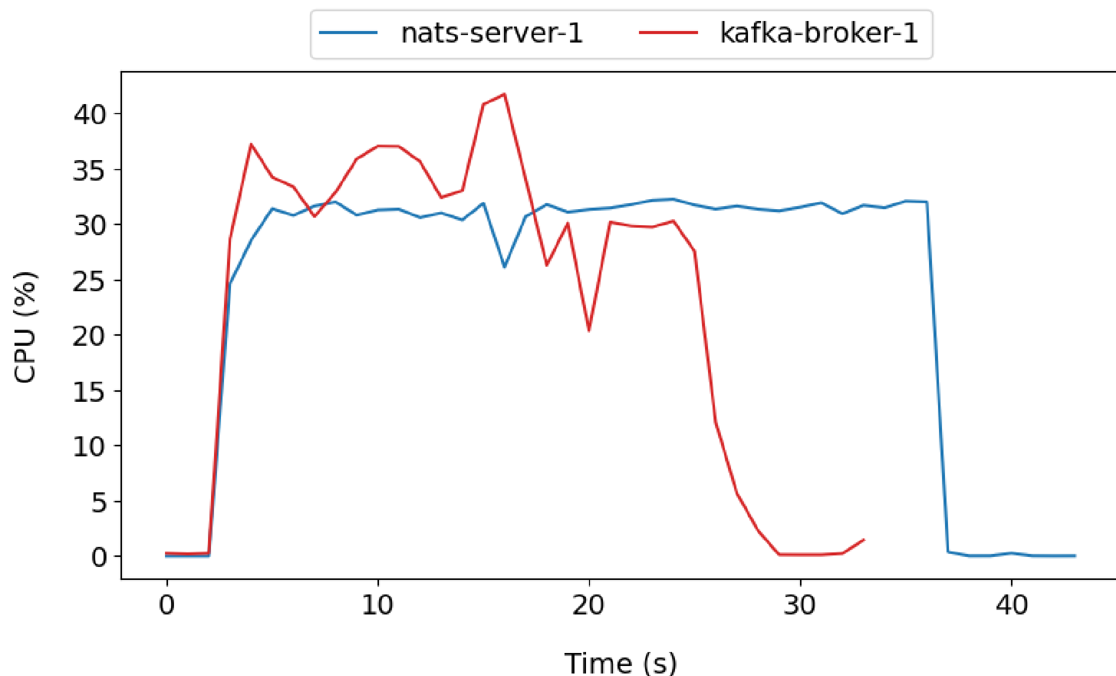

Figure 4.3: Single-Node CPU Usage

More significant differences can be seen in the three-node results shown in Figure

4.4. NATS's 95th percentile CPU usage was on average 0.8% higher across the nodes even though Kafka had a similar spike in the first half of the benchmark. There is also a larger difference in median CPU usage, where NATS had a 56.9% higher usage on average across the nodes compared to Kafka.

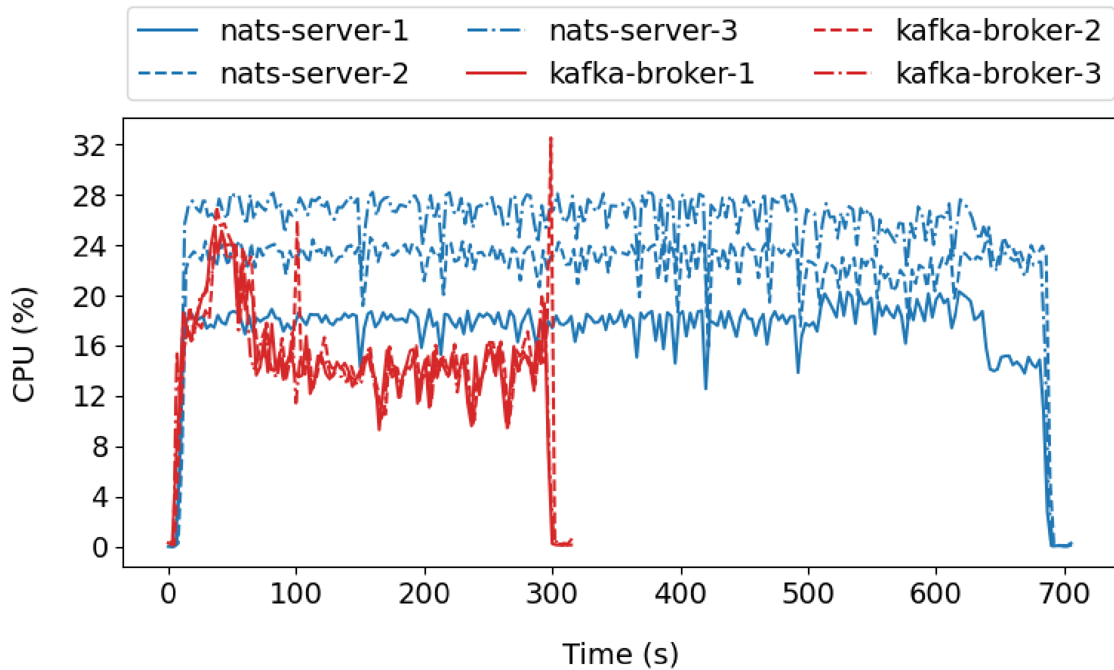

Figure 4.4: Three-Node CPU Usage

These results show that Kafka is able to process higher volumes of messages with lower overall CPU usage, especially in distributed environments. One likely reason for NATS's increased CPU usage is the lack of publisher batching, which leads to more CPU load per message. As a result, Kafka may offer lower CPU-related cloud computing costs or require less server capacity in on-premise setups.

Additionally, the distribution of CPU usage across NATS nodes varied significantly, while Kafka's brokers showed a more even distribution. This indicates that Kafka is better at distributing work across brokers, which helps handle bottlenecks in individual nodes more effectively. For NATS's benefit, its CPU usage is very

predictable based on these results, which can help with capacity planning.

4.5.4 Memory Usage

In single-node memory usage shown in Figure 4.5, NATS's maximum memory usage was 86.8% higher than Kafka's. However, this trend is not consistent with the three-node results shown in Figure 4.6. NATS's memory usage remained relatively low and stable, while Kafka continued to allocate increasingly more memory to finish the task faster. On average, Kafka's maximum memory usage across the brokers was 264% higher compared to NATS.

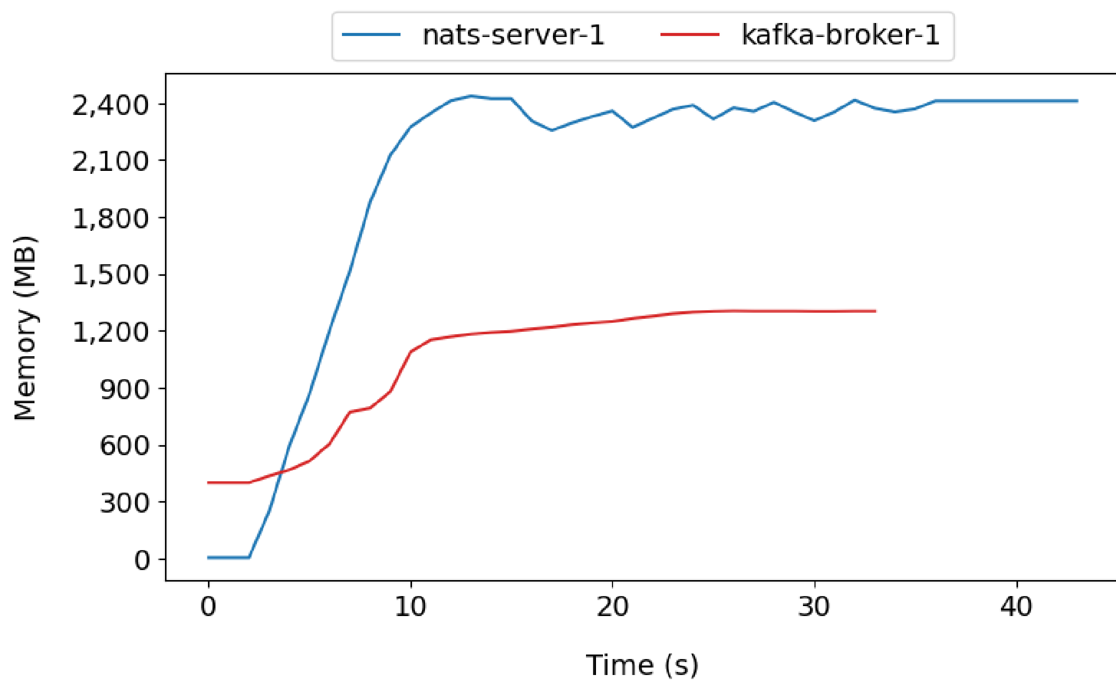

Figure 4.5: Single-Node Memory Usage

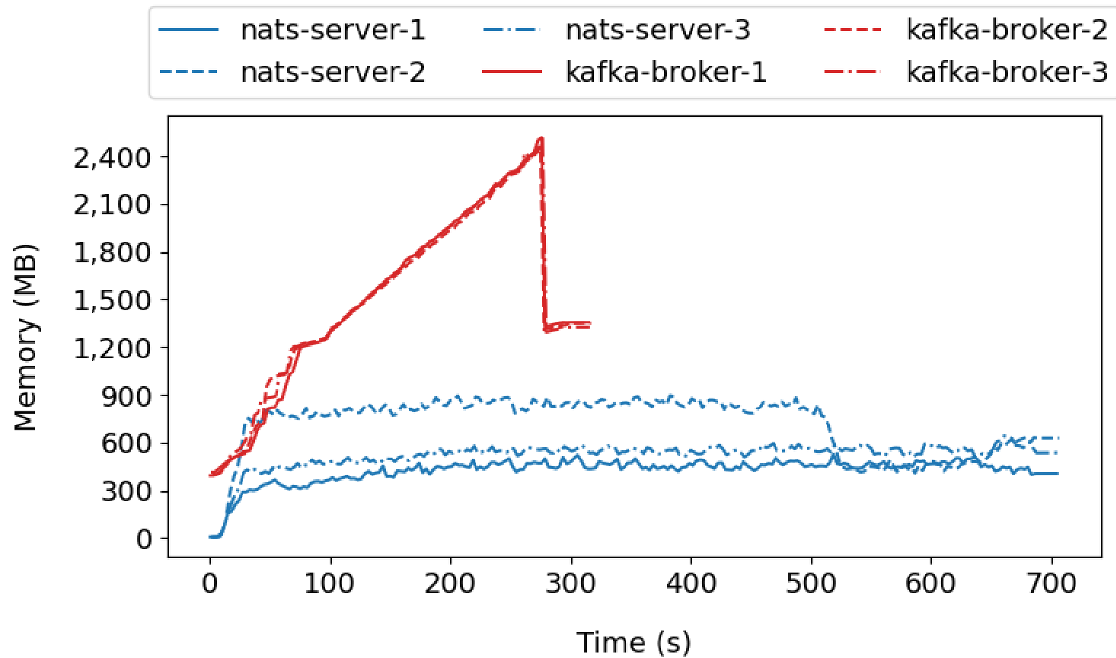

Figure 4.6: Three-Node Memory Usage

The results suggest that a single NATS node under high load can allocate a larger memory footprint, while coordination between nodes does not significantly increase memory usage in distributed environments. The results also suggest that distributed Kafka environments with large volumes of traffic can require large amounts of memory. These factors should be considered during capacity planning, particularly Kafka's high memory usage in distributed deployments.

The increased memory usage of NATS in the single-node benchmark cannot be justified, as it finished the task slower. Kafka's higher memory usage in the three-node benchmark is easier to justify, as it was able to finish earlier. Additionally, the distribution of memory usage across NATS server nodes is uneven, indicating the same imbalance in workload as seen in the CPU loads. In contrast, the memory usage across Kafka brokers shows no noticeable difference.

4.5.5 Latency

The latency differences between Kafka and NATS in Figure 4.7 and Figure 4.8 are primarily due to how each system handles sending messages. NATS published messages individually, where each publisher created the message, injected the timestamp header, sent the message, and then blocked until an acknowledgement was received. Kafka's producers buffered and batched messages before sending them to brokers, meaning that messages waited in the producer's memory for some time before being sent. This delay is visible in the results because the timestamp header was added when the message was created, not when it was sent. These results should not be interpreted as Kafka having high latency, but rather that NATS is able to achieve low latencies under the tested conditions.

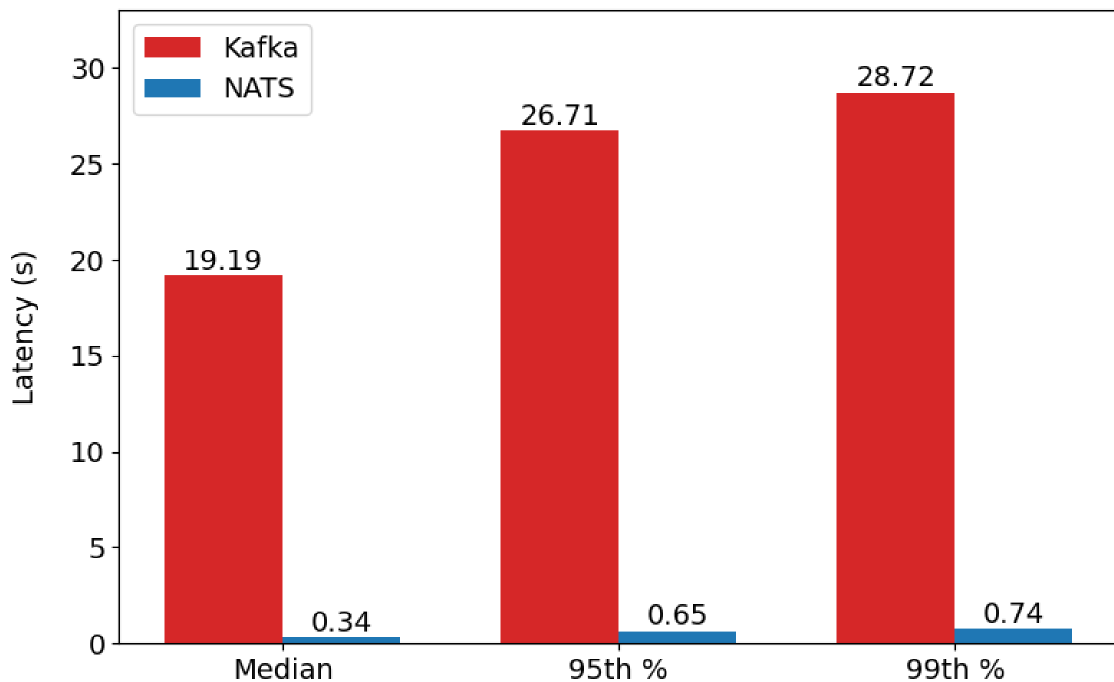

Figure 4.7: Single-Node Latencies

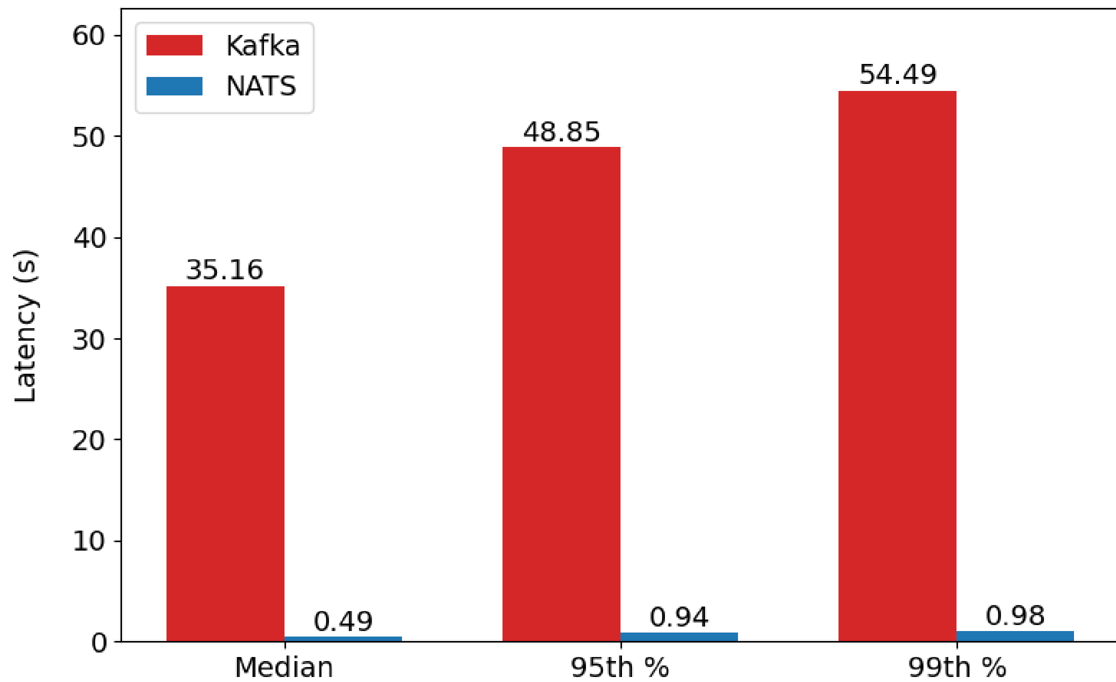

Figure 4.8: Three-Node Latencies

4.5.6 Throughput

Figure 4.9 shows the throughput of the single-node and three-node benchmarks. In the single-node setup, Kafka achieved 50.6% higher throughput compared to NATS, and in the three-node benchmark, this advantage increased to 130.7% higher throughput.

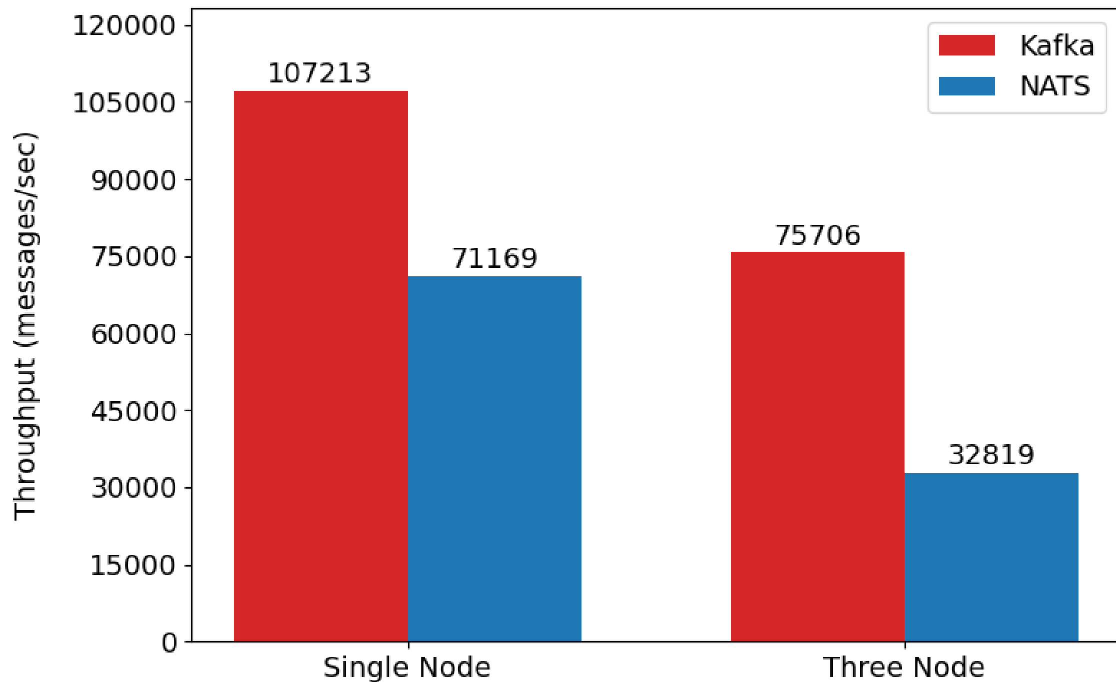

Figure 4.9: Throughput

These results highlight how much more effective Kafka is at ingesting large volumes of messages, even without network latencies where batching offers limited benefits. This gives Kafka a clear advantage in situations where messages are constantly created and sent. On the other hand, scenarios involving randomly timed single messages may benefit more from NATS, closing the gap in throughput.

4.5.7 Warmup Effects

Because Kafka is JVM-based, the same benchmarks were run a second time after the initial run to account for warmup effects. These results showed significant improvements. In contrast, NATS showed no differences between the initial run and a second run. In the second single-node benchmark for Kafka, throughput increased by 23.4%, and median latency dropped by 27.6%. While maximum memory usage rose by 14% due to the already allocated memory after the first run, median CPU

usage decreased significantly by 41%. Similar results were observed in the second three-node benchmark, where throughput increased by 8.9%, and median latency dropped by 22.5%. Maximum memory usage increased by an average of 11% across nodes, while average median CPU usage decreased by 6.2%.

These results highlight how Kafka benefits from runtime optimizations, reflecting real-world deployments more closely. After the warmup phase, it achieved higher throughput and lower CPU usage and latency. It is likely that Kafka would benefit even more in subsequent runs. However, Kafka also tended to allocate increasingly more memory over time. Overall, the observed differences in the results between Kafka and NATS mostly grew larger after the subsequent runs.

4.5.8 Impacts of Push Consumers and Async Publishing

To explore if NATS's synchronous publishing affected the results, asynchronous publishing with a window size of 16 messages was also tested. But managing an additional amount of in-flight requests compared to just one for each publisher resulted in worse throughput and latency. Because NATS doesn't support batching in the publisher side, each asynchronous publish still corresponds to an individual request to the server. Overall, NATS's blocking behavior did not negatively impact the results due to the use of lightweight virtual threads.

The same tests for NATS were also run using push consumers instead of pull consumers to identify any possible bottlenecks. In the single-node benchmark, push consumers achieved lower latencies of just one to two milliseconds due to their ability to deliver messages instantly without the need for polling. Throughput also increased by 21.2%, suggesting a minor bottleneck on the consumer side when using pull consumers. The elevated resource usage with push consumers supports this hypothesis. With push consumers, median CPU usage was 18.2% higher and median memory usage 12.5% higher, indicating the server was able to utilize more resources

to finish the task faster.

In the three-node benchmark, latencies were again in the range of three to nine milliseconds, but throughput increased by only 7.3%. This minor increase in throughput is reflected in the resource usage. On average, median memory usage was 6.1% higher across the nodes, while median CPU usage was 9.2% higher. These results indicate that while push consumers offer some performance gains, NATS can remain limited in its ability to intake higher volumes of messages using the current amount of publishers. To intake more messages, the amount of publishers has to be increased.

4.6 Evaluation of the Benchmark

This benchmark offered a controlled comparison of Kafka and NATS in identical container environments, but a few caveats remain. Because there is no network latency, the effects of Kafka's batching are understated compared to real-world deployments. The savings in network round trips would likely add to Kafka's advantage in throughput, while NATS would see a more significant performance hit. This compromises on the fairness of the benchmark.

Because both systems offer endless configuration and tuning options, especially Kafka, the configurations were kept constant and used mostly default values. While the effects of replication and cluster coordination were captured in another run in addition to a single-node setup, these results don't fully reflect real-world deployments, where configurations would be tuned to specific use cases and requirements. On the other hand, the ability to perform effectively with minimal configuration or tuning can be seen as an important feature. Additionally, the varying number of messages, clients, and communication channels in real-world deployments cannot be fully replicated in these benchmarks, so these results serve only as a reference point. These factors hinder the relevance of the benchmark.

Despite the limitations, this benchmark provided reproducible and verifiable results. Results remained consistent across multiple runs, and it was possible to confirm that each system successfully processed the full set of messages on every run. Both systems were allowed to utilize all available resources, with no artificial limitations on computing resources or configuration, adding to the fairness of the comparison.

5 Conclusions

This thesis set out to evaluate the suitability of NATS as a messaging middleware for event streaming in comparison to Kafka. Kafka is a widely used event streaming platform, but it requires careful management and configuration to reach its full potential, which is why NATS was explored as a more lightweight alternative. The differences were examined through a literature review comparing the features, and an empirical benchmark comparing key metrics and resource consumption.

The first research question (*How suitable is NATS for event streaming based on its features, and how does it compare with Kafka?*) was aimed at evaluating the capabilities of NATS relative to Kafka. This involved mapping, analyzing, and comparing the core features of both systems.

The results showed that Kafka's architecture is built more towards handling high volumes of data with strong ordering guarantees. In contrast, NATS offers greater versatility through features that can make it better suited for niche and specific use cases. It can be concluded that NATS can work as a viable alternative if requirements related to throughput, bandwidth efficiency, and strict message ordering are not the primary concern.

The second research question (*How does NATS compare to Kafka in event streaming in terms of performance and resource footprint?*) was aimed at measuring and comparing the throughput, latency, and resource efficiency of both systems. This involved using two server environments with different configurations to capture the

effects of both a simpler setup and a more complex distributed deployment with increased persistence.

The results added to the findings from the first research question by showing how the architectural differences between the systems translate into concrete differences in performance. Overall, Kafka demonstrated better throughput while also achieving better resource efficiency in most areas. This difference grew even larger after Kafka's runtime optimizations took effect. The results also highlight also how much more effective a single Kafka producer is compared to a NATS publisher. It can be concluded that Kafka is better suited for high throughput scenarios with a constant flow of messages. However, NATS still offers low latencies and the ability to handle a reasonable amount of throughput, while also being simpler to set up and requiring a smaller initial footprint.

Based on the results overall, there are no significant disadvantages in adopting NATS instead of Kafka. If NATS's throughput capacity meets the requirements, the main barrier might be Kafka's strong position in the ecosystem. This could impact migration efforts, since many third-party tools are well integrated into Kafka. For future research, exploring the compatibility and integration challenges when migrating existing workflows that depend on Kafka to NATS would be valuable. These results would help guide development of migration strategies, identify gaps in tooling, and minimize risks during migration.

Also, a more distributed benchmark using separate machines over the internet could provide better insights into how NATS handles incoming messages in real world network conditions. A greater number of concurrent publishers from multiple machines is suggested to fully capture NATS's ability to handle incoming load. Additionally, measuring client side behavior would provide insights into how resource intensive the clients are. Based on these results, it would be possible to determine the suitability of each client for low energy IoT environments with respect to energy

consumption and resource usage. Finally, it would be valuable to investigate how NATS's additional features like key value storage, service discovery, and object store compare to existing solutions. These results would help evaluate whether NATS can serve as a viable all in one solution for messaging, coordination and data storage, potentially reducing the need for external systems in distributed architectures.

References

[1] G. Banavar, T. Chandra, R. Strom, and D. Sturman, "A case for message oriented middleware", Sep. 1999, pp. 1–18, ISBN: 978-3-540-66531-1. DOI: 10.1007/3-540-48169-9_1.

[2] *What is event streaming?*, (Accessed 03/22/2025). [Online]. Available: https://www.ibm.com/think/topics/event-streaming.

[3] *Nats server*, (Accessed 01/20/2025). [Online]. Available: https://github.com/nats-io/nats-server.

[4] L. Magnoni, "Modern messaging for distributed sytems", *Journal of Physics: Conference Series*, vol. 608, no. 1, p. 012 038, Apr. 2015. DOI: 10.1088/1742-6596/608/1/012038.

[5] J. Yongguo, L. Qiang, Q. Changshuai, S. Jian, and L. Qianqian, "Message-oriented middleware: A review", in *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*, 2019, pp. 88–97. DOI: 10.1109/BIGCOM.2019.00023.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe", *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003, ISSN: 0360-0300. DOI: 10.1145/857076.857078.

[7] F. Araujo and L. Rodrigues, “On qos-aware publish-subscribe”, in *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, 2002, pp. 511–515. DOI: 10.1109/ICDCSW.2002.1030819.

[8] *Nats docs*, (Accessed 03/29/2025). [Online]. Available: https://docs.nats.io/.

[9] S. Patro, M. Potey, and A. Golhani, “Comparative study of middleware solutions for control and monitoring systems”, in *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, 2017, pp. 1–10. DOI: 10.1109/ICECCT.2017.8117808.

[10] P. Bellavista, A. Corradi, and A. Reale, “Quality of service in wide scale publishsubscribe systems”, *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1591–1616, 2014. DOI: 10.1109/SURV.2014.031914.00192.

[11] N. Ivaki, N. Laranjeiro, and F. Araujo, “A survey on reliable distributed communication”, *Journal of Systems and Software*, vol. 137, pp. 713–732, 2018, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.03.028.

[12] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann, “Performance evaluation of message-oriented middleware using the specjms2007 benchmark”, *Performance Evaluation*, vol. 66, no. 8, pp. 410–434, 2009, Selected papers of the Fourth European Performance Engineering Workshop (EPEW) 2007 in Berlin, ISSN: 0166-5316. DOI: https://doi.org/10.1016/j.peva.2009.01.003.

[13] P. Tran, P. Greenfield, and I. Gorton, “Behavior and performance of message-oriented middleware systems”, in *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, 2002, pp. 645–650. DOI: 10.1109/ICDCSW.2002.1030842.

[14] C. Fernando, *Designing Microservices Platforms with NATS: A modern approach to designing and implementing scalable microservices platforms with*

NATS messaging. Packt Publishing, 2021, ISBN: 9781801076623. [Online]. Available: https://ieeexplore.ieee.org/document/10162727.

[15] P. Sommer, F. Schellroth, M. Fischer, and J. Schlechtendahl, "Message-oriented middleware for industrial production systems", in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, 2018, pp. 1217–1223. DOI: 10.1109/COASE.2018.8560493.

[16] J. Kreps, "Kafka : A distributed messaging system for log processing", 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:18534081.

[17] G. Fu, Y. Zhang, and G. Yu, "A fair comparison of message queuing systems", *IEEE Access*, vol. 9, pp. 421–432, 2021. DOI: 10.1109/ACCESS.2020.3046503.

[18] *Kafka design overview*, (Accessed 03/16/2025). [Online]. Available: https://docs.confluent.io/kafka/design/.

[19] S. S P, V. Kumar C, and S. P. P, "Application monitoring and telemetry analytics", in *2024 7th International Conference on Circuit Power and Computing Technologies (ICCPCT)*, vol. 1, 2024, pp. 1559–1565. DOI: 10.1109/ICCPCT61902.2024.10673415.

[20] *Kafka*, (Accessed 04/01/2025). [Online]. Available: https://github.com/apache/kafka.

[21] *Kafka 4.0 documentation*, (Accessed 03/29/2025). [Online]. Available: https://kafka.apache.org/documentation/.

[22] M. Kleppmann and J. Kreps, "Kafka, samza and the unix philosophy of distributed data", 2015. DOI: 10.17863/CAM.33349. [Online]. Available: https://www.repository.cam.ac.uk/handle/1810/286031.

[23] A. B. A. Alaasam, G. Radchenko, and A. Tchernykh, "Stateful stream processing for digital twins: Microservice-based kafka stream dsl", in *2019 International Multi-Conference on Engineering, Computer and Information Sciences*

(SIBIRCON), 2019, pp. 0804–0809. DOI: 10.1109/SIBIRCON48586.2019.8958367.

[24] *Jmx exporter*, (Accessed 04/22/2025). [Online]. Available: https://prometheus.github.io/jmx_exporter/.

[25] T. Aung, H. T. Zaw, A. H. Maw, and M. T. Mon, "Comprehensive analysis: Monitoring apache kafka with grafana, jmx exporter, and prometheus", in *2024 5th International Conference on Advanced Information Technologies (ICAIT)*, 2024, pp. 1–6. DOI: 10.1109/ICAIT65209.2024.10754944.

[26] *Prometheus*, (Accessed 04/22/2025). [Online]. Available: https://prometheus.io/.

[27] *Nats weekly #33*, (Accessed 03/25/2025). [Online]. Available: https://www.synadia.com/newsletter/nats-weekly-33.

[28] *Nats weekly #19*, (Accessed 04/28/2025). [Online]. Available: https://www.synadia.com/newsletter/nats-weekly-19.

[29] *Prometheus-nats-exporter*, (Accessed 04/22/2025). [Online]. Available: https://github.com/nats-io/prometheus-nats-exporter.

[30] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey", *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015. DOI: 10.1109/TKDE.2015.2427795.

[31] *Kafka-nats-bench*, (Accessed 05/09/2025). [Online]. Available: https://github.com/ninosalonen/kafka-nats-bench.

[32] J. von Kistowski, J. Arnold, K. Huppler, K.-D. Lange, J. Henning, and P. Cao, "How to build a benchmark", Feb. 2015. DOI: 10.1145/2668930.2688819.

[33] W. Hasselbring, "Benchmarking as empirical standard in software engineering research", in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '21, Trondheim, Norway: Association for Computing Machinery, 2021, pp. 365–372, ISBN: 9781450390538. DOI: 10.1145/3463274.3463361.

[34] H. Wu, Z. Shang, and K. Wolter, "Performance prediction for the apache kafka messaging system", in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 154–161. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00036.

[35] G. van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks", *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1845–1858, 2020. DOI: 10.1109/TPDS.2020.2978480.

[36] *Storage drivers*, (Accessed 04/22/2025). [Online]. Available: https://docs.docker.com/engine/storage/drivers/.

[37] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers", in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[38] *Docker compose*, (Accessed 04/22/2025). [Online]. Available: https://docs.docker.com/compose/.

[39] *Amazon ebs pricing*, (Accessed 05/20/2025). [Online]. Available: https://aws.amazon.com/ebs/pricing/.

Appendix A Use of Generative AI

Generative AI was used in two ways while writing this thesis. First, it was partially used with searching for and evaluating possible solutions for measuring the required metrics from Kafka and NATS. However, none of the suggested tools were suitable, so a custom solution was built. For this custom solution, generative AI supported the development of the test environment by providing coding assistance and helping troubleshoot errors. All fixes and suggestions were evaluated, and only those that improved the solution were accepted. Some fixes were further enhanced manually when necessary.

Second, generative AI was used to improve text quality and language. These improvements were made without changing the content or meaning of the original text. A common prompt used was the following: *"Point out any obvious grammar and language mistakes in the following text without changing the original meaning or structure. Ignore minor stylistic preferences."* The suggested fixes were then evaluated and applied only if they provided value and improved the quality of the text.