Umeå University
Report

May 22, 2024

UMEÅ UNIVERSITY

Evaluating the Performance of Serialization Protocols in Apache Kafka

Tobias Myastovskiy

Tobias Myastovskiy
Spring 2024
Degree Project in Computing Science and Engineering, 30 credits
Supervisor: Alexandre Bartel
External Supervisor: Andreas Grahn & Marcus Henriksson
Examiner: Henrik Björklund
Master of Science Programme in Computing Science and Engineering, 300 credits

Abstract

In the context of data–intensive applications, efficient data serialization is essential for maintaining high performance and scalability. This thesis investigates the impact of different serialization protocols on the latency and throughput in Apache Kafka, a widely used distributed streaming platform. Given the diverse array of serialization protocols, this study focuses on four prevalent ones: Apache Avro, Protocol Buffers (Protobuf), JSON, and MessagePack. These protocols were selected based on their widespread use in academic research and industry and their varying approaches to balancing human readability, efficiency, and performance.

JSON, the most commonly used serialization protocol in many systems, is a baseline for comparison in this study. While JSON offers ease of use and broad compatibility, it may not be optimal in terms of speed and data size efficiency. This research aims to determine whether alternative serialization protocols can improve performance.

This research utilized a testing framework involving two distinct types of tests: batch processing and single message processing. Each test type consisted of 1,048,575 records and was applied across three different data sizes, 1,176 bytes, 4,696 bytes, and 9,312 bytes, to evaluate how the data size impacts serialization and deserialization times, total execution times, throughput, and latency. The throughput is measured in records per second (rps).

The throughput results indicate that MessagePack achieves a two–time higher throughput than JSON. The batch–processing results from lowest to highest size show 34,254 rps vs 14,243 rps, 7,377 rps vs 3411 rps, and 3,802 rps vs 1784 rps. The single–message results show 29,212 rps vs. 14,126 rps, 8,350 rps vs. 3,344 rps and 3,781 rps vs. 1,803 rps. Protobuf showed the highest throughput for the smallest tested data size at 36,945 rps for batch–processing and 36,364 rps for single message processing. Avro showed a slight edge over JSON regarding throughput but was less significant than MessagePack. All the protocols were faster than JSON regarding serialization speeds, the quickest one being Protobuf.

Regarding latency, Protobuf consistently achieved the lowest median latencies across all test sizes in batch processing, recording 38.97 ms, 57.41 ms, and 63.14 ms for increasing record sizes, whereas JSON showed higher latencies of 77.59 ms, 72.60 ms, and 78.09 ms. In single-message tests, Protobuf also displayed the lowest median latency at 1.68 ms for the smallest size, significantly outperforming JSON's 7.94 ms. Interestingly, for the record size of 4,696 bytes, JSON exhibited the lowest median latency at 3.76 ms. Avro presented the lowest median latency for the largest size at 2.71 ms, compared to JSON's 4.18 ms.

The results indicate that migrating from JSON to MessagePack or Protobuf (for the lowest size) will increase throughput by twofold. Protobuf enhances latency metrics across all tested sizes in batch–processing scenarios, making it a convincing choice for systems prioritizing rapid data handling. For single–message tests, Protobuf is recommended for the smallest data size, while Avro offers advantages for the largest data size.

Acknowledgements

Thanks to Alexandre Bartel for being a good supervisor and asking critical questions. Thanks to my external supervisors, Andreas Grahn and Marcus Henriksson, for providing the necessary resources for my thesis and supporting me along the journey.

Contents

List of Figures

List of Algorithms

List of Tables

1 Introduction

In the realm of data–intensive applications, ensuring high performance presents numerous challenges. These applications often require the processing and transmitting of vast amounts of data at high speeds, necessitating efficient data handling to avoid bottlenecks. Among the critical factors influencing such systems' performance is the data serialization method.

Serialization is the process of converting data structures into a format that can be easily stored or transmitted and subsequently reconstructed. Efficient data serialization is crucial for optimizing the throughput and latency of data transmission systems. The choice of serialization protocol can significantly impact the overall system performance, affecting key metrics such as serialization and deserialization times, total execution times, and the efficient use of network and storage resources.

The external partner of this thesis, currently uses JSON as their serialization protocol within their Kafka–based data transmission system. JSON's human–readable format offers ease of use and widespread support but may not provide the best performance regarding speed and compactness of data. This context underpins the research questions of this thesis, which aim to explore whether alternative serialization protocols could offer significant performance improvements.

This master thesis investigates the performance implications, specifically latency and throughput of different serialization protocols within Apache Kafka [5], a popular distributed streaming platform. Given the variety of serialization protocols available, each with its own strengths and weaknesses, the protocol choice can significantly impact the overall system performance. This study focuses on four widely used serialization protocols: Apache Avro, Protocol Buffers (Protobuf), JSON, and MessagePack [4, 16, 20, 27]. These protocols were selected based on their prevalence in academic research and industry applications and their varying approaches to balancing human–readability, efficiency, and performance.

1.1 Research Questions

By conducting comprehensive performance tests under different conditions, this thesis aims to provide insights into how these serialization protocols affect key performance metrics in Apache Kafka, and whether alternative serialization methods could yield better performance outcomes. The study seeks to answer the following research questions:

RQ1. How do different serialization protocols compare regarding latency and throughput in Apache Kafka?

RQ2. Are the benefits significant enough to consider migrating from JSON to another serialization protocol?

1.2 Thesis Outline

The thesis is structured by first providing the necessary theory of the components part of this thesis, presented in Section 2, followed by Section 3, presenting existing literature related to the topic. After the theory is presented, Section 4, describes the design of the conducted tests and the hardware specifications, and the results are presented in Section 5. The results are discussed in Section 6 regarding the research questions, and in Section 7, a conclusion is drawn from the results.

2 Background

This section provides the essential background information required to understand the context and core concepts of the thesis. The content outlines the theoretical information about distributed systems, Apache Kafka and Serialization, and briefly describes Amazon Kubernetes Service.

2.1 Distributed Systems

Distributed systems have been given several definitions across academia, generally characterized as a network of independent computers appearing as a coherent system to its users. These systems achieve coherence through synchronization methods and communication protocols that ensure all components work together seamlessly [26, Ch. 1.1]. For instance, algorithms such as Lamport timestamps or vector clocks are often employed to manage the order of events and maintain consistency across the network [26, Ch. 6.2]. This complexity allows distributed systems to efficiently handle large–scale, data–intensive applications that a single computer system could not manage. However, this complexity introduces several operational challenges, including the difficulty of keeping a consistent time across all machines. This is known as the global clock problem. Since each machine has its own clock, getting them to agree at the same time is tricky and requires special coordination methods. Other issues include keeping data consistent across different locations and handling failures when parts of the system stop communicating. These problems make designing and operating distributed systems complex, often requiring careful planning and management [26].

Network Time Protocol (NTP) addresses the global clock problem by synchronizing the clocks of computers across a network, ensuring that all components within a distributed system agree on the same time. NTP achieves this by using a *server* and *client* concept, where the server is a source of time information, and the client is a part of the distributed system that requests and receives time information to find out its skew.

The client starts by sending a request containing the timestamp of when it left the client (T1). The server receives the message from the client and records the received timestamp (T2). A response packet containing the timestamps T1 and T2 is then assembled, and a response packet is destined for the client (T3). Finally, the client receives the response from the server and records the received timestamp as (T4).

Having all four timestamps, the client can calculate the round–trip delay (the time it takes for a message to travel to the server and back to the client) and the time offset (the clock difference between the client's clock and the server's clock). The formulas are,

Round–trip time: $\delta = (T4 - T1) - (T3 - T2)$

Time offset: $\theta = \frac{(T2-T1)+(T3-T4)}{2}$

The time offset calculation is based on the assumption that the message travels the same way under the same circumstances on the way back from the server to the client as from the client to the server, essentially assuming that the transmission times are the same in both ways [18].

2.1.1 Throughput

Throughput is a performance metric in distributed systems, representing the rate at which data can pass through a system in a given amount of time. In other words, it is a measure of efficiency. Throughput is measured in data bytes per second (BPS), or transactions per second (TPS). To calculate the throughput the total work size, denoted W_{Total}, is divided by the total time taken to transmit the data, denoted T_{Total},

$$Throughput = \frac{W_{Total}}{T_{Total}}$$

2.1.2 Latency

Latency is another performance metric in distributed systems, representing the time it takes for data to pass from one point to another on a network. A longer delay results in a higher latency. There are several causes for increased latency, such as long geographical distances, slow server processing, different transmission mediums, high data volume, and the number of network hops [2, 10]. To measure the latency all the mentioned latency increases have to be considered, provided in,

$$Latency = Propagation + Transmission + Processing + Queuing \tag{1}$$

Propagation delay measures the time it takes for a bit to travel through the network medium. The bits travel either in the form of electromagnetic signals in the case of copper cables or a laser in the case of fiber optics. The propagation delay is influenced by the physical distance between the endpoints. To calculate the propagation delay, the physical distance denoted D is divided by the speed of the transmission medium denoted V_{medium}, resulting in the formula,

$$PropagationDelay = \frac{D}{V_{medium}}$$

Transmission delay measures the time required to push all the bits onto the network link. It depends on the packet size (bits) and the bandwidth (bits per second) of the network link. To calculate the Transmission delay, the packet size, denoted S_{packet}, is divided by the bandwidth, denoted R, giving the formula,

$$TransmissionDelay = \frac{S_{packet}}{R}$$

Processing delay is the time it takes for routers, switches, and other networking hardware to process a packet and determine its next hop. It can vary depending on the complexity of the network hardware or software.

Queuing delay is the time data spends in queues before being processed and transmitted by the networking hardware. Expressing a formula for queuing delays is challenging, as it depends on the traffic intensity and the queue–managing policies.

Finally, the network latency is the sum of these delays (1), plus any additional delays introduced by the network, such as retransmission delays in case of packet loss [1, 19, 33].

2.1.3 Publish/Subscribe Paradigm

Publish/subscribe (pub/sub) systems represent a paradigm shift in how applications communicate in distributed environments. Distinct from the conventional *remote invocation* protocol and *point–to–point*, the pub/sub architecture offers a temporally and referentially decoupled system. A temporal coupling means that the processes that communicate with each other have to be up and running, resembling a phone conversation with people that have to be on each end of the line for the phone call to happen. A referential coupling means that the process must have an exact address to refer to when communicating, much like when sending mail, the sender must know the recipient's address. Due to the nature of its architecture, a pub/sub system does not need to know its recipient or if the recipient is on the end of the line. The pub/sub system achieves this form of decoupling by utilizing a "shared memory space", known as middleware, between the different components in the distributed systems[26, Ch. 2].

A middleware (in distributed systems) is a software layer between the operating system of the computers that are a part of the distributed system and the application(s) [26, Ch. 1-2]. A middleware can be seen as a resource manager for a distributed system, offering applications to share and deploy resources over a network. Middleware can also provide security services, application communication, and accounting services, only to name a few, much like an operating system. In the case of a pub/sub system, the middleware serves as a *Message Oriented Middleware* (MOM). A MOM provides a loosely coupled system where components communicate asynchronously, reliably, and efficiently.

An important aspect of pub/sub systems is how subscribers describe the data they want to receive. There are two types of pub/sub systems: a topic–based pub/sub system and a content–based pub/sub system. A content–based pub/sub system allows subscribers to specify more detailed criteria for the messages they want to receive based on the content of the messages. Instead of subscribing to a topic, subscribers define filters that the messages must have to receive them. However, this comes at the cost of complexity in the pub/sub system and performance

On the other hand, in a topic–based pub/sub system, subscribers express their interest in receiving messages by subscribing to a predefined topic. These topics act as categories publishers use to classify the messages they publish. When a message is published on a topic, the message broker ensures that the message is delivered on that specific topic, ready for subscribers to consume the data [26, Ch. 2.1]. A visualization of a topic–based pub/sub system can be seen in Figure 1.

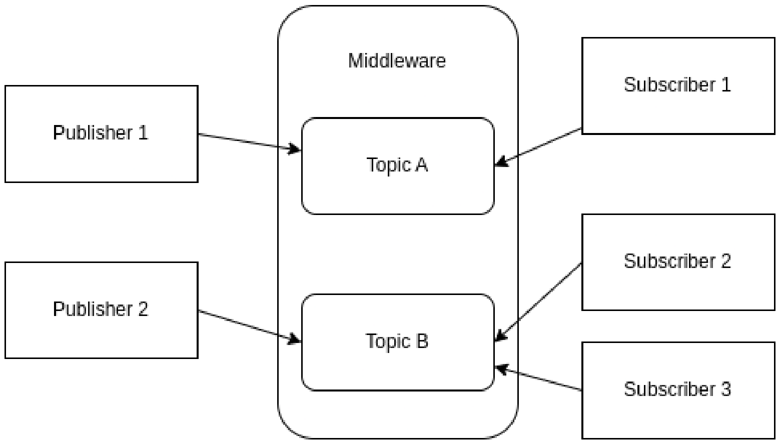

Figure 1: An example of what a topic–based publish/subscribe system can look like.

The figure shows the temporal and referential decoupling of the producer and consumer, as well as how the publisher, message broker, and subscriber communicate.

Several frameworks are based on the pub/sub architecture, the more popular ones being Apache Kafka [5] and RabbitMQ [37]. Both of them have their advantages and disadvantages, but when it comes to throughput and latency, Kafka is the better option in terms of throughput [14, 38].

2.2 Apache Kafka

Apache Kafka is a distributed event (data) streaming platform, based on the topic–based pub/sub architecture, designed for high throughput, fault–tolerance, and scalability [5, 40]. In pub/sub systems there are publishers and subscribers, in the Kafka architecture they are referred to as clients, called producers and consumers, respectively. A Kafka server is called a broker and is designed to operate as part of a cluster of many brokers. A cluster usually has a replication of brokers in the same cluster, which provides redundancy of the messages.

A data unit within Kafka is called a message, which Kafka sees as an array of bytes, not concerned about its contents. In Kafka, messages bound to the same topic are transmitted in batches rather than one at a time, to improve the throughput. Transmitting the messages one at a time would result in an excessive overhead. The batches can be compressed, providing a smaller batch size to transfer, resulting in efficient data transmission and storage [40, pp. 4-5].

The topics in Kafka are broken down into partitions, a subset of a topic's data. In Kafka, There is no guarantee that messages arrive in order across a topic with multiple partitions, only within a single partition. Partitioning topics allows Kafka to achieve redundancy and scalability by allowing partitions for a topic to be spread over different Kafka brokers on different machines. This enables a topic to scale horizontally across multiple Kafka brokers, providing performance increases compared to a single broker. Copies of partitions can also be stored across Kafka brokers, allowing for redundancy [40, pp. 5-6]. Figure 2 shows a topic with 4 partitions, receiving different amounts of data in each partition.

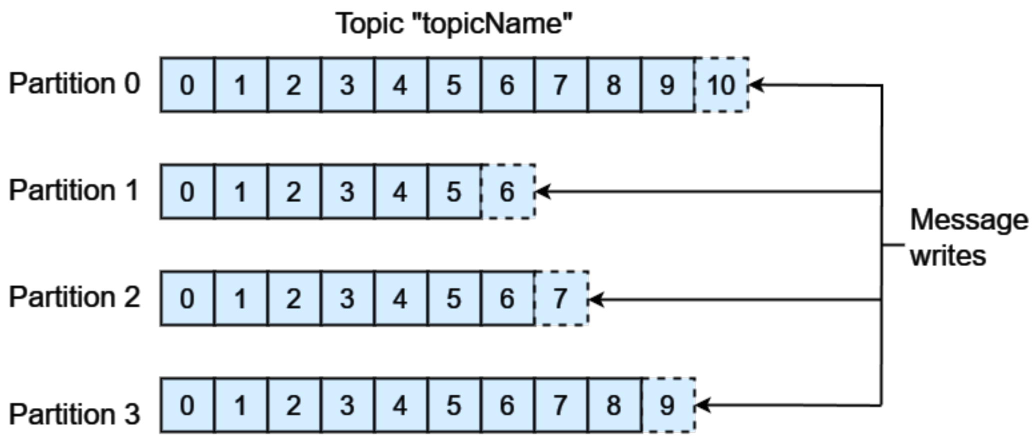

Figure 2: A representation of a topic with 4 partitions.

Producers create new messages and publish them on a specified topic on the Kafka broker. The messages are by default distributed evenly across the partitions in a Round–Robin fashion [40, pp. 68-69]. However, there are instances where a producer might target messages to specific partitions leading to unfair message distribution among paritions.

Consumers can subscribe to one or more topics and read the messages in the order they were produced for each partition. The consumer maintains a record of the messages it has already processed by tracking the offset of those messages. The offset is stored in the metadata of each message. The current offset is typically stored in Kafka, allowing consumers to fail, restart, and continue from where they left off. Consumers operate as members of a consumer group. A consumer group consists of one or more consumers, collaborating to consume from the same topic. Consumers are mapped to one or more partitions, and each mapping is called an *ownership* of the partition. There can only be one owner of a partition at a time, ensuring a partition is only consumed by one consumer. This setup allows for horizontal scaling, enabling consumers to process high volumes of messages. In the event of a consumer failure, the other group members will redistribute the affected partitions among themselves, ensuring continuous processing [40, pp. 6-7]. Figure 3 illustrates how a consumer group consisting of three consumers is reading messages from a topic with 4 partitions (same topic as in Figure 2).

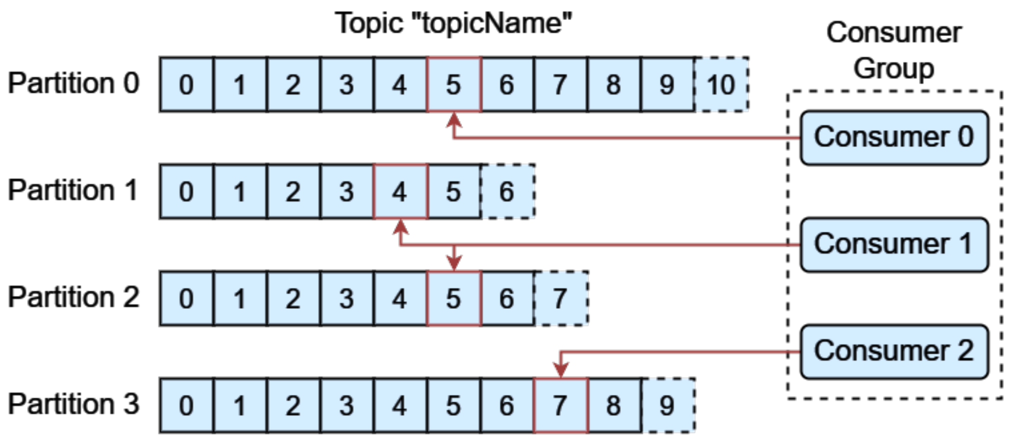

Figure 3: A consumer group with 3 consumers reading from a topic with 4 partitions.

The Kafka broker is responsible for delivering the messages received from the producer to the consumers. The broker achieves this by receiving messages from the producer, assigning offsets to them, publishing them on the partitions, and responding to fetch requests made by the consumer. Brokers are intended to function collectively within a cluster. One broker will be elected to act as a cluster *controller*, overseeing various administrative tasks, such as assigning partitions to brokers and monitoring broker failures. Each partition within the cluster can be owned by one broker only, referred to as the *leader* of the partition. Furthermore, partitions can be replicated across multiple brokers, with these brokers serving as *followers* of the partition as illustrated in Figure 4 [40, pp. 8-9]. Replication of partitions ensures message redundancy, facilitating followers to take over leadership in case of broker failures.

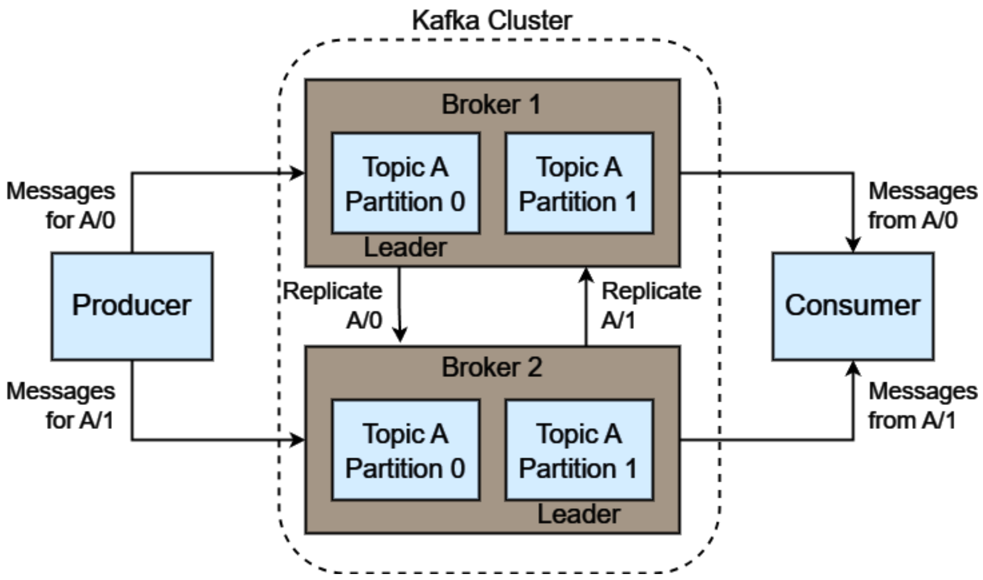

Figure 4: Replication of partitions across 2 brokers.

Now, with an overview of the Kafka infrastructure, one can configure Kafka based on their use case. There are several configurations of Kafka, starting with compression.

Compression

Compression is designed to reduce the size of files and data streams, enabling more efficient storage and faster transmission. Compression algorithms minimize the bits required to represent information. There are two main types of compression: lossless and lossy. Lossless compression reconstructs (decompresses) the original data perfectly. Lossy compression achieves greater size reduction at the cost of data loss [35, Ch. 1].

In Kafka, the default settings leave messages uncompressed. However, Kafka does support four compression algorithms `snappy`, `gzip`, `lz4` and `zstd`. Each has its benefits and drawbacks. The primary distinction lies in their compression efficiency, which impacts the message size and the compression time. Higher compression rates lead to smaller messages but demand extra CPU usage, bandwidth, and time to compress.

Table 1: Compression performance comparison between 4 compression algorithms [39].

Compression Type	CPU Usage	Compression Speed	Bandwidth Usage
`Gzip`	Highest	Slowest	Lowest
`Snappy`	Medium	Moderate	Medium
`Lz4`	Low	Fastest	Highest
`Zstd`	Medium	Moderate	Medium

Table 1 compares four compression types across various metrics. Gzip stands out for having the lowest bandwidth at the cost of the slowest compression speed and highest CPU usage. On the contrary, Lz4 is notable for its fast compression speed and low CPU usage, but at the cost of the highest bandwidth usage. Both Zstd and Snappy provide a balance, with medium values of the 3 measured metrics, suggesting a compromise between efficiency and resource utilization.

Batch Size

The batch size parameter denotes the maximum size in bytes a batch can reach before sending it to the Kafka broker. A larger batch size allows more messages to be batched together before being sent off which can increase throughput at the cost of latency and memory consumption (on the producer) [40, pp. 59].

Linger Time

The linger time denotes the maximum time the batch will wait to accumulate messages before being sent to the Kafka brokers. The linger time parameter complements the batch size parameter. If the linger time is set to 0, the batch will wait until it matches the batch size parameter. If both the batch size and linger time parameters are set, the batch will be sent once one of the parameters is satisfied [40, pp. 59].

Acknowledgement

The acknowledgment parameter, known as “acks”, dictates the reliability of a message delivery. This parameter determines how many acknowledgments the producer requires from the brokers for a message to be considered successfully sent. There are several options for the parameter to be configured with, each having an implication on reliability, latency, and throughput [40, pp. 55].

`acks=0`, when set to 0, the producer will not wait for any acknowledgments from the brokers, resulting in the lowest latency at the cost of reliability.

`acks=1`, when set to 1, the producer expects one acknowledgment from the leader broker to ensure the delivery of the message to the broker. The message can still be lost if the broker crashes after acknowledging the message but before replicating it to other brokers. Overall this option is a good balance between latency and reliability.

`acks=all (acks=-1)`, is the safest mode. It requires acknowledgments from all in–sync replicas before considering the message as successfully sent. This setting ensures a message is not lost if one broker fails and the other in–sync replicas are still functional. This is the most reliable option at the cost of latency and throughput.

2.3 Azure Kubernetes Service

Kubernetes is an open–source, container orchestration platform and simplifies the deployment, management, and operation of containerized applications. Azure Kubernetes Service (AKS), provides a managed Kubernetes environment that streamlines these processes by automating numerous management aspects, such as cluster deployment, scaling, and updates [29]. Applications like Apache Kafka are well suited to be deployed on a Kubernetes cluster due to its scalable nature, allowing Kubernetes to manage demands on the application dynamically, providing fault–tolerance, redundancy, high availability, and scalability.

Having applications deployed on AKS implies they are situated within one or multiple Azure data centers. Given that the Azure public cloud is widely utilized and has a wide range of tenants who simultaneously use their resources, the performance of applications, particularly in terms of latency may be impacted by how other tenants utilize the Azure resources. The rented resources are not guaranteed to lie within the same data rack. They could be spread across different racks within the data center. This can lead to increased network activity from other tenants during peak hours, which can result in congestion, leading to higher latency for packet delivery. This is especially critical for latency–sensitive applications, where data processing and message delivery delays can affect overall system performance. Additionally, the multi–tenancy model of the public cloud can lead to "noisy neighbor" issues, where other tenants' heavy usage of CPU, memory, or I/O resources can adversely impact the performance of your deployed applications. Azure's underlying infrastructure tries to mitigate this by providing isolation and load–balancing strategies, but during times of high demand, these mechanisms might be stressed [9].

2.4 Serialization

Serialization is the process of converting data structures or data objects into a format that can be stored or transmitted. The serialized object can later be reconstructed to its original state [41]. This technique has evolved alongside technological advancements, addressing the growing need for efficient data exchange and storage.

Serialization protocols range from binary representations, prioritizing efficiency and performance, to human–readable formats, which enhance accessibility and ease of understanding. Binary protocols use schemas to define the structure of how the data is converted into a binary format and how to revert it to its original state. Examples include MessagePack, Protocol Buffers and Apache Avro, which offer compactness and speed, making them suitable for high–performance computing and network communications. Conversely, human–readable formats like JSON are ideal for configurations, debugging, and scenarios where data needs to be easily read and modified by humans.

Key–value pairs play a crucial role in serialization, providing a flexible and intuitive way to model data. In JSON and MessagePack, data is often organized using key–value pairs, simplifying data mapping to objects in programming languages. Similarly, Protobuf and

Avro schemas define keys (fields) and their corresponding values, ensuring structured data representation and consistency during serialization and deserialization.

The choice of serialization protocol depends on the application's specific requirements, including network bandwidth and processing speed. Criteria such as size, speed, readability, and ease of use will be considered to evaluate and compare serialization protocols thoroughly. These factors help to determine the most suitable protocol for a given application, balancing performance with accessibility and security requirements.

Understanding the distinct characteristics and use cases of various serialization protocols is crucial for selecting the most appropriate one for a given application. The following subsections will delve deeper into four widely used serialization protocols: Apache Avro, Protocol Buffers (Protobuf), MessagePack, and JSON. Each headline will provide an overview of the protocol, its key features, and its advantages and disadvantages.

JSON - JavaScript Object Notation

JavaScript Object Notation (JSON) is a text–based serialization protocol for data interchange. It is designed with human readability in mind, making it easy to read and write. JSON structures data as key–value pairs, which allows for flexible and intuitive data representation. It supports various data types, including strings, numbers, arrays, objects, and booleans, making it exceptionally versatile for representing complex data hierarchies.

JSON's simplicity and ease of use have contributed to its widespread adoption, especially in web development, where it facilitates seamless data exchange between servers and client–side applications. Unlike binary serialization formats, JSON does not require a predefined schema, simplifying data interchange and reducing overhead. This schema–less nature allows for dynamic and flexible data structures, though it can also lead to potential issues with data validation and consistency.

Despite its advantages, JSON has some limitations. It is less efficient in terms of storage and processing speed compared to binary formats like Protocol Buffers and Avro. JSON's text–based nature can lead to larger file sizes and slower parsing times, which may not be suitable for high–performance applications with strict efficiency requirements.

JSON is supported natively by most modern programming languages, making it a convenient choice for interoperability in diverse environments [7, 20].

MessagePack

MessagePack is a binary serialization format similar to JSON but designed to be more compact and efficient. Unlike JSON, which is text–based and easily readable by humans, MessagePack encodes data in a binary format, resulting in smaller message sizes. This compactness makes it more efficient for network transmission and storage, as it reduces the amount of data needed to be transmitted or stored. However, it is important to note that while MessagePack is more compact than JSON, its size is not significantly smaller and is still considerably larger compared to other binary serialization formats like Apache Avro and Protocol Buffers.

MessagePack structures data as key–value pairs, similar to JSON, and supports a wide range of data types including integers, floats, strings, arrays, and maps. This versatility allows it to represent complex data structures efficiently. Additionally, MessagePack's compatibility with many programming languages enhances its integration capabilities.

Despite its advantages, the binary format of MessagePack makes it less human–readable, which can complicate debugging and manual data inspection. However, this trade–off is often acceptable in scenarios where performance and efficiency are prioritized over readability [8, 22, 27].

Protocol Buffers

Developed by Google, Protocol Buffers (Protobuf) is a binary serialization format for structured data. Protobuf is designed to be small and fast, making it optimal for network transmission, especially where bandwidth and performance are critical. It requires a schema to define the structure of the data before it can be serialized or deserialized [16].

Protobuf uses a compact binary format, which results in smaller message sizes compared to text–based formats like JSON. This compactness makes Protobuf highly efficient for storage and network transmission, reducing the amount of data that must be handled. The predefined schema ensures data is consistently structured, allowing for efficient parsing and validation.

Protobuf is highly efficient for applications requiring compact and fast data serialization, such as remote procedure calls (RPC). The compact binary format, robust schema support, and high performance make it ideal for critical bandwidth and speed scenarios. However, Protobuf may not be suitable for handling extremely large data sets exceeding a few megabytes, as it assumes that entire messages can be loaded into memory at once, potentially leading to high usage. Additionally, Protobuf messages are not inherently compressed, which can be a drawback for certain types of data. Protobuf also lacks optimal efficiency for scientific and engineering applications involving large multi–dimensional arrays. Furthermore, Protobuf is not a formal standard, limiting its use in environments requiring adherence to standardized protocols, and its support is limited in non–object–oriented languages commonly used in scientific computing [16].

While Protobuf is highly efficient, it is less human–readable than text–based formats like JSON, which can complicate debugging and manual data inspection. However, tools and libraries can convert Protobuf messages to human–readable formats for easier debugging.

Compared to other binary serialization formats like Apache Avro and MessagePack, Protobuf tends to produce smaller message sizes and offers better performance. This makes it particularly suitable for environments where efficiency and speed are paramount.

Apache Avro

Apache Avro is a binary serialization format developed within the Apache Hadoop project, focusing on data serialization for big data applications. Like Protocol Buffers, Avro is designed to be compact and efficient, facilitating fast data serialization and deserialization, making it well–rounded for high–volume data exchange and storage scenarios. A distinctive feature of Avro, in contrast to Protobuf, is its schema–on–read capability, where the schema is dynamically resolved at the time of data reading, rather than strictly requiring both producer and consumer to have access to the same schema at write time. This makes Avro particularly suitable for scenarios involving schema evolution, as it simplifies the integration of new schema versions.

However, Avro may not be the best fit for applications requiring extremely small message sizes or ultra–fast serialization speeds, as formats like Protobuf may perform better in these areas. Moreover, while compact, Avro's binary format, is not human–readable, potentially complicating debugging and manual data inspection. Finally, Avro might not be well–suited for real–time applications or systems with strict latency requirements where minimal processing delay is critical [4].

3 Related Work

In [17] a detailed examination of the performance of various serialization protocols within a publish-subscribe middleware context for Multi-Processor System-on-Chips (MPSoCs) was carried out. The study evaluates four major serialization formats: JSON, XML, Protocol Buffers (Protobuf), and Apache Avro, focusing on their serialization and deserialization efficiencies. The methodology involves a comparative analysis using the same sets of data across all protocols to ensure consistency in the evaluation. The experiments are designed to measure both the time taken for serialization and deserialization processes and the size of the serialized data, which is crucial for optimizing communication and storage in constrained environments.

The results from the study highlight significant differences in performance between the protocols. JSON and XML, while human-readable, incur higher overheads in terms of time and data size due to their text-based nature. In contrast, Protobuf and Avro, both binary formats, demonstrate superior performance with faster serialization and deserialization times and more compact data sizes. Among these, Avro provides an optimal balance of speed and data efficiency, partly due to its schema–evolution capabilities that suit the needs of dynamic systems within MPSoCs well.

This paper mainly evaluates the performance of serialization protocols in MPSoCs, which is not the domain of this thesis. However, it discusses the performance of serialization protocols, which indirectly impact the domain, providing insight and expectations of how the protocols behave.

The configuration parameters in Apache Kafka play a significant role in throughput and latency performance. A benchmarking analysis presented on Red Hat's developer platform [6] investigates various Kafka configurations to understand their impact on producer performance, exploring configurations such as batch size, linger time, and acknowledgment. These configurations are critical for optimizing throughput and reliability in Kafka producers.

The results from the study indicate that adjustments in these configurations can lead to significant differences in performance outcomes. For example, increasing the batch size and adjusting the linger time increased message throughput, allowing more messages to be sent in a single batch before dispatch. The study also evaluates the impact of different acknowledgment settings, which balance throughput efficiency and data reliability. However, the benchmarking analysis was conducted with only the "Snappy" compression algorithm.

Similarly, a study conducted in 2017 about identifying the impact of different parameter settings on Apache Kafka [24], concluded that the batch size impacted the throughput negatively as the number of Kafka brokers grew in the cluster without having any message replication active. The researchers suspected the decrease in performance had to do with Kafka's internal synchronization and the underlying network. This study has several limitations, such as not disclosing the linger time and only providing charts that show latency and throughput data without the message replication option active.

Moreover, a recent study on the compression algorithms in Kafka [32] provides an in–depth evaluation of how the supported compression algorithms, namely GZIP, Snappy, LZ4, and ZSTD, affect the throughput and latency. The experimental setup involved measuring both message delivery latency and throughput under controlled conditions to ensure an unbiased comparison of the compression types.

The findings from this study reveal significant variations in performance across the different compression algorithms. While all compression options reduced the overall message size, leading to efficiencies in network usage, they exhibited varying impacts on throughput and latency. ZSTD emerged as the superior choice as the message size increased, providing the best impact on throughput and latency, closely followed by Snappy. This makes ZSTD and Snappy attractive compression options for Kafka deployments where performance and efficiency are essential.

In the comparative study of Kafka and RabbitMQ [12], both systems are evaluated across several dimensions critical to distributed pub/sub architectures, particularly focusing on throughput and fault tolerance. The study revealed that while Kafka provides superior throughput due to its distributed log system, RabbitMQ offers lower latency per message, suited for scenarios requiring rapid message delivery rather than large–volume processing. Furthermore, Kafka's robust scalability makes it ideal for applications needing extensive data logging and stream processing capabilities. It contrasts with RabbitMQ's flexible routing and multi–protocol support that accommodate complex transactional systems. Importantly, the paper identifies optimal use cases for each system based on their architectural strengths and operational characteristics. It suggests Kafka for high–throughput needs and RabbitMQ for scenarios requiring sophisticated message routing and immediate delivery.

4 Methodology

This Section presents the methodology used to evaluate the impact of serialization protocols on throughput and latency in Apache Kafka. The content outlines the choice of serialization protocols, the platform setup, specifying the specifications of the virtual machines, and the Kafka setup, followed by the experimental setup. The experimental setup presents the test suites, the producer and consumer configurations in relation to Kafka, and their underlying algorithms for the production and consumption of messages.

4.1 Serialization Protocols

This section presents the motivation for the choice of serialization protocols and information about each protocol's compactness.

4.1.1 Choice of Serialization Protocols

The serialization protocol sphere is vast and encompasses various formats, making the selection of suitable protocols for evaluation a crucial step in this study. The selection criteria were based on throughput, latency, and prevalence considerations. Protocols with small message sizes, particularly binary formats, were prioritized to address throughput and latency concerns. Additionally, it is valuable to compare human–readable frameworks with those optimized for latency and throughput.

After extensive research on serialization protocols, the following protocols were chosen for this study: Apache Avro, Protocol Buffers (Protobuf), JSON, and MessagePack.

Apache Avro and Protobuf were selected among binary serialization protocols due to their frequent appearance in both academic research [25, 34, 42] and industry discussions, including web articles and blog posts [15, 21, 23]. These protocols are renowned for their efficiency and compactness, making them suitable for high–performance applications.

For the human–readable condition, JSON was chosen due to it being used by my external partner and due to its widespread usage and recognition in research [25, 34, 42] and various articles [15, 23]. JSON is known for its lower serialized data size compared to XML and YAML, which contributed to the exclusion of the latter two formats from this study [13, 43].

MessagePack was selected as a lightweight alternative to JSON. It offers similar capabilities but in a more space–efficient manner, making it an interesting candidate for comparison in terms of both human readability and performance [20, 22, 27].

4.1.2 Compactness

Presented in Section 4.3, the tests were conducted on three distinct record sizes 1,176 bytes, 4,696 bytes, and 9,312 bytes. The serialized data size varies depending on which protocol it is serialized with. Table 2 presents the serialized sizes of the original data (1,176 bytes, 4,696 bytes, and 9,312 bytes). It shows that across all three record sizes, Protobuf serializes the data into the smallest size, followed by Avro, MessagePack, and JSON, in that order.

Table 2: The serialized sizes of the original data of sizes (1,176 bytes, 4,696 bytes, and 9,312 bytes)

Serialization Protocol	**1176 B**	**4696 B**	**9312 B**
`Avro`	239 B	1063 B	2093 B
`JSON`	1178 B	5954 B	11885 B
`MessagePack`	1019 B	5261 B	10512 B
`Protobuf`	181 B	825 B	1630 B

An example of what sample data might look like when serialized with the different protocols can be found in Appendix A. The data presented in the Appendix is just sample data to illustrate what the same data might look like when serialized with different protocols and should not be confused with the data used in the experiments. The sample data shows that Avro is serialized into a smaller size than Protobuf, which appears larger in this specific instance. However, it will produce smaller sizes as the data size increases as illustrated in Table 2.

4.2 Setup

This section outlines the setup used for the experimental evaluation of serialization protocols in Apache Kafka and is divided into two detailed sections. Section 4.2.1 provides information on the computational resources from Azure used in the tests, detailing the specific hardware configurations. Section 4.2.2 describes the Kafka configuration, including topics, replication factors, partition settings, and broker deployment.

4.2.1 Platform Setup

The tests were conducted using resources from the Azure cloud platform, provided by the external thesis partner. The underlying hardware for the Azure Virtual Machines (VMs) includes one of the following processors: 3rd Generation Intel Xeon Platinum 8370C (Ice Lake), Intel Xeon Platinum 8272CL (Cascade Lake), or Intel Xeon Platinum 8168 (Skylake). The VM instance type deployed was "Fsv-2", equipped with 32 virtual CPU cores and 64 GB of RAM. Notably, the Fsv-2 series features a high CPU-to-memory ratio, making it well-suited for batch–processing tasks [30].

4.2.2 Kafka Setup (Client & Broker)

Broker

The Kafka architecture was configured with one distinct topic for each serialization protocol, each with a replication factor 1. That means if the broker on which the partition resides fails, the data is lost if the broker can not be recovered. Each topic is configured with a partition factor of 3, meaning each can have three parallel consumers. Three brokers were deployed to support this setup, meaning each broker will handle one partition independently.

Clients

In this case, the clients are the producer and consumers. For the client setup, one producer and three consumers were chosen. There is no educated reason for choosing three consumers other than the assumption that one consumer might be a bottleneck for the whole system.

4.3 Experiments

As discussed in Section 3, Kafka outperforms other message–oriented middleware with its superior throughput capabilities, particularly in batch–processing scenarios. However, batch–processing can lead to higher latencies. To evaluate this trade–off, two distinct types of tests were conducted. The first test involved batching messages before sending, to assess the impact on throughput. The second test involved sending messages individually to evaluate the effect on latency.

The external partner, provided the test data consisting of 1,048,575 records. Three distinct record sizes were used for each tested serialization protocol: 1,176 bytes, 4,696 bytes, and 9,312 bytes. By varying the record sizes for each test, the aim is to evaluate how different serialization protocols and Kafka's performance are influenced as the record size increases.

All the tests were written in Python 3.10.12. The libraries used for the serialization protocols are presented in Table 3.

Table 3: The Python libraries used for the serialization protocols.

Serialization Protocol	Python Library
`Apache Avro`	Fastavro [3]
`JSON`	Built–in JSON library [36]
`MessagePack`	MessagePack for Python [28]
`Protocol Buffers`	Protocol Buffers for Python [16]

To prevent any cold runs or inconveniences from affecting the timings, the serialization, deserialization, producer, and consumer timings were averaged on three runs. The runs were conducted during office hours, in the public cloud, prone to "noisy neighbor" issues (Section 2.3). Running the tests during office hours gives more realistic results for the external partner of this thesis, but does not represent a general case where the test would be conducted in a controlled environment, mitigating the resource and network utilization from third parties.

The next headlines describe the configurations of the different test methods and provide specific details on the consumer and producer.

Test 1 (Batch–Processing)

For the batch-processing tests, optimal settings for batch size and linger time were essential. Based on recommendations from [11], setting the batch size within the 100,000 to 200,000 bytes range and a linger time between 10 and 100 ms can significantly enhance throughput when properly tuned. Accordingly, a batch size of 100,000 bytes and a linger time of 100 ms were selected. The acknowledgment setting was configured to 1, to balance throughput with the small risk of message loss. The compression algorithm “Snappy” was employed for all batch–processing tests due to its efficacy in achieving high throughput with minimal impact on latency, as discussed in Section 3.

Test 2 (Single Message Processing)

Test 2 consisted of sending the messages one by one, removing the batch size and linger time from the producer configuration. The compression algorithm remained Snappy, and the acknowledgment configuration was set to 1.

Producer

The producer was configured to measure two critical timings: the time required to serialize the data and the total time to transmit all 1,048,575 records. Each transmitted message includes a timestamp, representing the time elapsed since the UNIX epoch (January 1, 1970, at midnight UTC/GMT) [31], which is used by the consumer to calculate the time in transit for that message. The time–taking of the producer started as the producer started looping through the data records.

The core functionality of the producer, which computes these timings, is encapsulated in the algorithm provided (see Algorithm 1). This algorithm initializes the start time measurement, iterates over records fetched from a database, builds the data object, serializes the data object, and appends the serialization time to a list for later analysis. The timestamp for each message is adjusted and synchronized with a central NTP (Network Time Protocol) server, ensuring that the timing data for each record is as precise as possible. After processing all records, a “completion” message is sent to indicate that all messages have been successfully transmitted. Subsequently, the total transmission time is recorded, measuring the producer’s efficiency and overall performance. The producer algorithm is mostly the same for Test 1 & Test 2.

The algorithm constructs a data object because serialization requires data to be in a key–value format. Only the values are typically retrieved when fetching data from the database. These values must then be explicitly assigned to corresponding keys to prepare them for serialization. This key–value mapping is essential for serialization, as it structures the data so that the serialization protocol can accurately and efficiently process.

Algorithm 1 Producer serialization and transmission loop.

```
start_time ← time.time()
for record in cur.fetchall() do                                    ▷ Fetch from DB
    data_dict ← data_dict_template.copy()
    for $i$ ← 0 to len(record) do
        if isinstance(record[i], datetime) then
            data_dict[keys[i]] ← record[i].isoformat()
        else
            data_dict[keys[i]] ← record[i]
    ser_start ← time.perf_counter()
    serialized_data ← serialize(data_dict)
    serialization_times.append(time.perf_counter() - ser_start)

    timestamp ← int(time.time() + time_offset)
    producer.produce(
        topic_name,
        value = serialized_data,
        callback = delivery_report,
        timestamp = timestamp)
total_time ← time.time() - start_time

for part ← 0 to 2 do              ▷ Notify the consumers that transmission is complete
    producer.produce(
        topic_name,
        value = 'PRODUCER_DONE',
        key = 'PRODUCER_DONE',
        partition = part,
        callback = delivery_report)
```

Consumer

The consumer measures the time it takes to consume all the messages on the topic, the deserialization time, and latency. The latency is measured by recording the time a message or a message batch was consumed, looping through the batch (in the case of the batch–processing test), and subtracting the recorded time of consuming the message/message batch from each message timestamp, measuring the duration each message spent in transit. The start time is recorded as the first message is consumed.

The initial functionality of both consumers is encapsulated in the algorithm provided (see Algorithm 2). This algorithm begins by polling for individual messages with a specified timeout of 1 s. Upon receiving a message, it first records the start time to measure the entire consumption duration of all 1,048,575 records. Secondly, it records the consumption time representing the timestamp the message was consumed to calculate the latency. Lastly, it deserializes the message and measures the time taken for the deserialization process.

After consuming the first message, the consumer proceeds to another very similar loop, which can be viewed as the "main" loop, where the rest of the consumption happens. The loop differs briefly between the batch consumer and the single message consumer, the major difference being that the batch consumer tries to consume messages in batches of up to 5000 messages while the single consumer only consumes one message at a time. The measurement of the latency and deserialization time is the same in both algorithms. Upon identifying a specific key in the last message of the batch, indicating the completion of all messages sent by the producer, the consumer performs a final commit, captures the end time to assess the total duration of the consumption process, proceeds with database insertions, and exits. The "main" loop of the batch consumer and the single message consumer can be seen in Algorithm 3 & 4 respectively.

The deserialization time–taking includes starting a high–resolution timer right before the deserialization process and stopping it immediately after to capture the exact duration of this operation. These timings are collected and stored for later analysis. After processing the message, the consumer commits the message to Kafka, marking it as processed, and then breaks from the loop to handle new incoming messages in batches.

Algorithm 2 Initial consumer consumption and deserialization loop.

```
while true do
    msg ← consumer.poll(1.0)
    if msg is None then
        print("Waiting for messages...")
        continue
    else
        start_time ← time.time()
        con_time ← int(time.time() + time_offset)
        latency.append(con_time - msg.timestamp())
        des_start ← time.perf_counter()
        deserialize(msg.value())
        deserialization_times.append(time.perf_counter() - des_start)
        consumer.commit(msg)
        break

```

Algorithm 3 Main consumer consumption and deserialization loop for the batch processing test.

```
while true do
    msg_batch ← consumer.consume(num_messages=5000, timeout=0.1)
    con_time ← int(time.time() + time_offset)
    if msg_batch then
        if (msg_batch[len(msg_batch) - 1].key() is not None and
            msg_batch[len(msg_batch) - 1].key() == 'PRODUCER_DONE') then
            if (len(msg_batch) - 1) > 1 then
                calculate_data()
            consumer.commit()
            insert_into_db()
            sys.exit(0)
        for msg in msg_batch do
            latency.append(con_time - msg.timestamp())
            des_start ← time.perf_counter()
            deserialize(msg.value())
            deserialization_times.append(time.perf_counter() - des_start)
        consumer.commit()
        end_time ← time.time()
```

Algorithm 4 Main consumer consumption and deserialization loop for the single message processing test.

```
while true do
    msg ← consumer.poll(1.0)
    con_time ← int(time.time() + time_offset)
    if msg is not None then
        if (msg.key() is not None and
            msg.key() == 'PRODUCER_DONE') then
            end_time ← time.time()
            consumer.commit()
            insert_into_db()
            sys.exit(0)
        latency.append(con_time - msg.timestamp())
        des_start ← time.perf_counter()
        deserialize(msg.value())
        deserialization_times.append(time.perf_counter() - des_start)
        consumer.commit()
```

In summary, Algorithms 1, 2, 3, 4, represent a general algorithm for all the serialization and deserialization protocols. There is no general serialization/deserialization function covering all the different serialization protocols. The setup and initialization of the programs differ depending on the protocol in question, as Avro and Protobuf are schema–based protocols and require a different setup from JSON and MessagePack. The algorithms are similar in the two test cases and prevalent in the consumer. For the batch–processing test, the "main" loop consumes messages in batches, which it then loops through to measure latency and deserialization time, while for the single message test, it only consumes one message, measures the relevant timings, and deserializes the message, quickly heading on to consuming the next message.

5 Results

This section presents the results of the tests described in Section 4.3. Firstly, the batch–processing results are presented in Section 5.1, followed by the results of sending one message at a time presented in Section 5.2.

The results are measured and compared for each serialization protocol. They are presented as serialization & deserialization times, total execution times, the serialization & deserialization impact on the total time, the throughput, and the latency.

The metric, "impact of the serialization protocols on the total time" presents the serialization and deserialization times as proportions of the total time for each protocol and size, presenting the duration of the serialization and deserialization times compared to the total time.

The measurements, serialization & deserialization times, and the impact of the serialization protocols on the total time do not have a strong correlation to the research questions (Section 1.1) due to them not directly relating to the throughput or latency. However, they display the efficiency of the serialization and deserialization processes, which directly impacts the total execution time, which in turn impacts the throughput. The throughput is calculated by dividing the total number of records (1,048,575) by the execution times of the producer and consumers, resulting in a measurement expressed in records per second.

5.1 Batch Processing

Serialization & Deserialization Times

The serialization and deserialization times for the various protocols are aggregated and presented in Figure 5. Across all three record sizes, Protobuf consistently showed the shortest times, with times of 6.271 seconds for 1176 bytes, 28.145 seconds for 4696 bytes, and 58.176 seconds for 9312 bytes, clearly outperforming the other protocols. This was followed by MessagePack, which recorded 8.088 seconds, 40.322 seconds, and 92.207 seconds respectively, for the same record sizes. Avro and JSON exhibited longer times, with JSON being the least efficient across all sizes.

In a detailed comparison, Protobuf was found to be approximately five times faster than JSON for the record sizes of 1176 bytes (6.271s vs. 34.164s), 4696 bytes (28.145s vs. 142.501s), and 9312 bytes (58.176s vs. 297.227s). MessagePack, the second fastest protocol, consistently performed within a competitive range, never more than twice as slow as Protobuf. For instance, at 9312 bytes, MessagePack's time of 92.207 seconds is roughly 1.58 times that of Protobuf but still considerably faster than Avro and JSON.

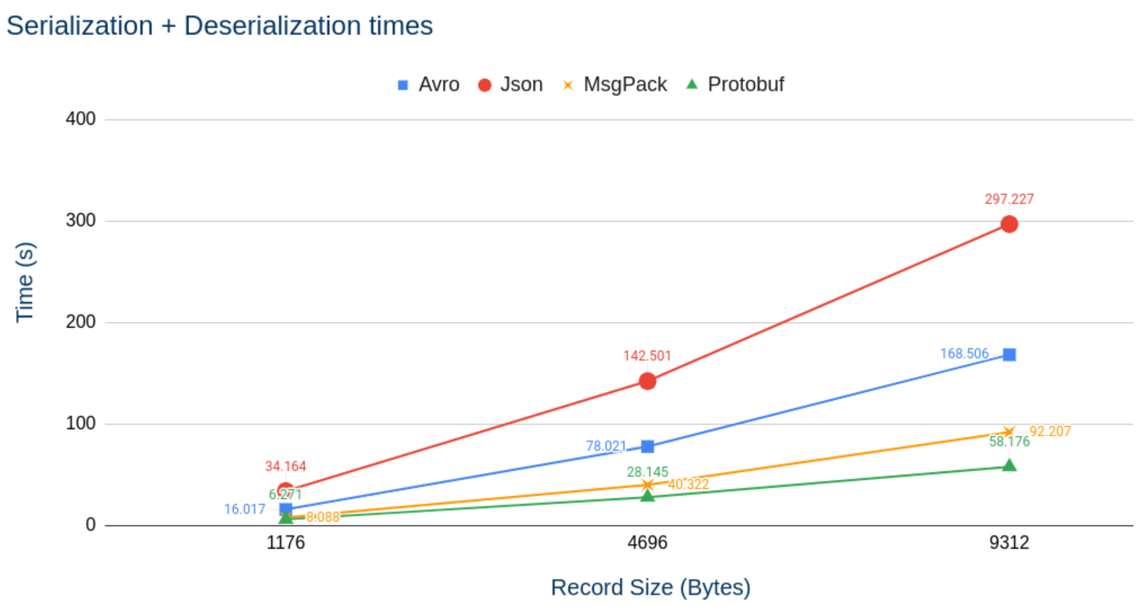

Figure 5: The aggregated serialization and deserialization times for each serialization protocol for the batched tests. The X-axis represents the size of each record, the Y-axis represents the time in seconds, and each line represents a serialization protocol.

Execution Times

The total execution times for the producer and the consumers are aggregated and presented in Figure 6. At the smallest record size, Protobuf performs the best, completing in 28.382s, outperforming MessagePack by a slight margin of 8%. However, as the record size increases, Protobuf's execution time increases rapidly, allowing MessagePack and Avro to overtake it. MessagePack emerges as the best performer at the two larger sizes completing at 142.143 seconds for the record size of 4696 bytes and 275.779 seconds for 9312 bytes, followed by Avro. When comparing the best–performing protocol (MessagePack) with the worst-performing protocol (JSON), MessagePack is consistently at least twice as fast across all record sizes.

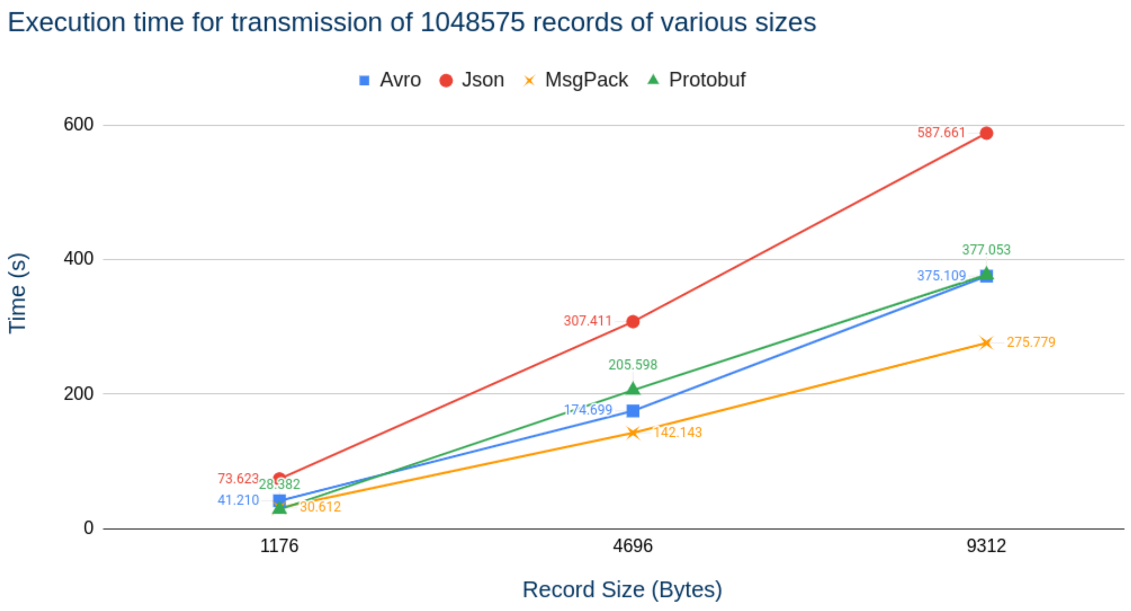

Figure 6: Aggregated producer and consumer times for the batch processing test for each protocol and each size. The X-axis represents the size of each record, the Y-axis represents the time in seconds, and each line represents a serialization protocol.

Serialization Impact on Total Time

The impact of serialization times on the total execution time varies by protocol and record size, as illustrated in Figure 7. Generally, serialization times increase with the record size across all protocols. However, Protobuf exhibits the opposite trend, where serialization times decrease as record sizes increase.

For Avro, the proportion of time spent on serialization and deserialization remains relatively consistent across the larger record sizes, with an increase from the smallest record size, accounting for 38.9% at 1176 bytes, increasing slightly to 44.7% at 4696 bytes, and then to 44.9% at 9312 bytes. JSON shows a slight increase in serialization and deserialization proportion as record sizes grow, starting from 46.4% at 1176 bytes, maintaining at 4696 bytes, and increasing to 50.6% at 9312 bytes. MessagePack shows an increasing trend, with 26.4% at 1176 bytes, increasing to 28.4% at 4696 bytes, and 33.4% at 9312 bytes. Notably, Protobuf shows the most decrease in serialization and deserialization times as a proportion of total time, starting at 22.1% for 1176 bytes, decreasing to 13.4% at 4696 bytes, and increasing slightly to 15.4% at 9312 bytes.

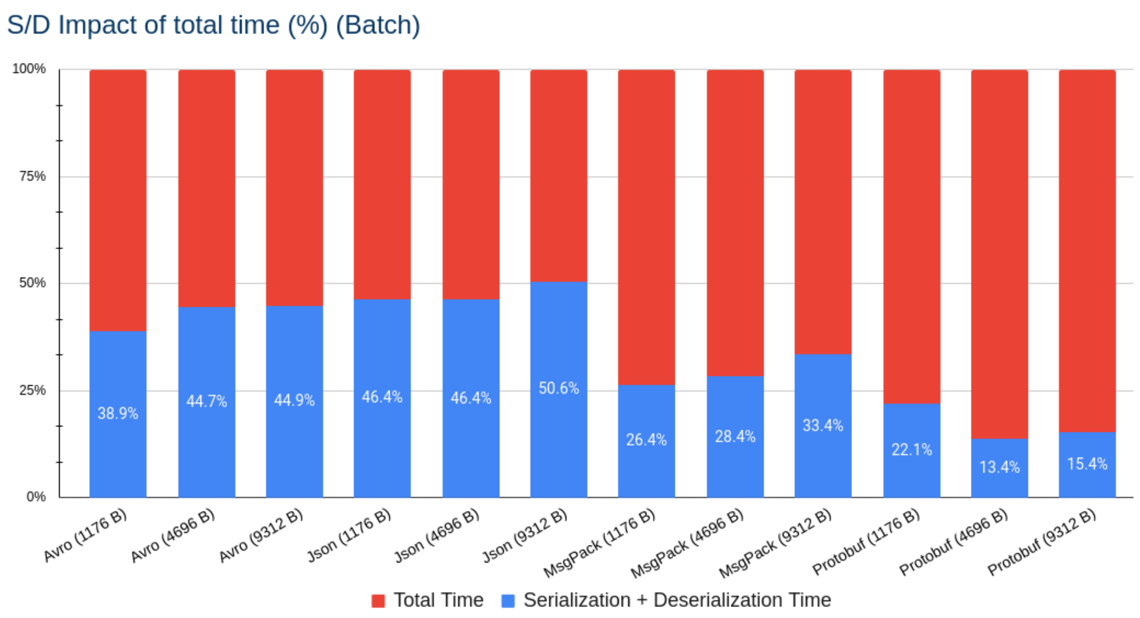

Figure 7: Serialization time and deserialization times over the total time for each protocol and size for the batch processing test. The blue areas represent the percentage of how much of the time was spent serializing or deserializing. The red area represents the Total time, excluding the serialization times.

Throughput

Figure 8 displays the throughput rates for the different serialization protocols. For the smallest record size (1176 bytes), Protobuf achieves the highest throughput at 36,945 records per second, slightly outperforming the second-best protocol, MessagePack, at 34,254 records per second. At the record size 4,696 bytes, MessagePack takes the lead with 7,377 records per second, ahead of Avro's 6,002 records per second, and even further beyond Protobuf's 5,100 records per second. At the largest size of 9312 bytes, MessagePack still leads with 3,802 records per second, which is 36% higher than Avro's 2,795 records per second. Protobuf shows a throughput of 2,781 records per second. At the same time, JSON performs the worst across all record sizes, with a throughput of 1,784 records per second at 9312 bytes, marking it as at least two times less efficient than MessagePack across the evaluated record sizes.

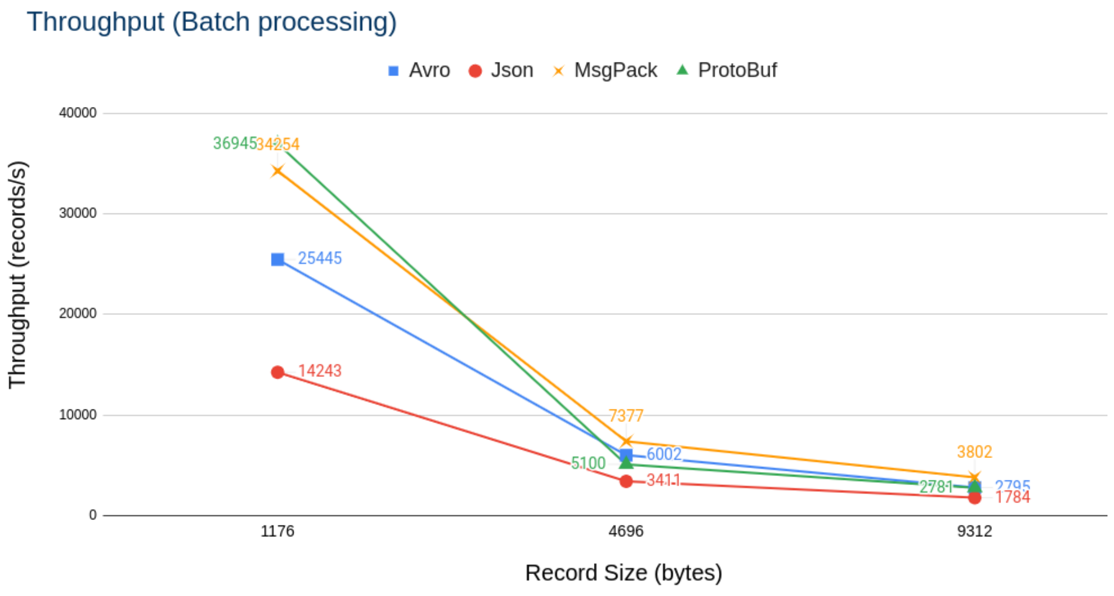

Figure 8: Throughput for the different serialization protocols for the batch–processing tests. The X-axis represents the size of each record, the Y-axis represents the time in seconds, and each line represents a serialization protocol.

Latency

The latency results are presented in three different graphs, each representing a record size. The record sizes are 1176 bytes, 4696 bytes, and 9312 bytes, and the latencies for the record sizes are presented in Figures 9, 10 & 11 respectively.

For the smallest record size of 1176 bytes, Protobuf demonstrates the lowest median latency at approximately 38.97 ms, followed closely by MessagePack, with a median latency of 45.75 ms. Avro shows a higher latency of 52.94 ms, and JSON has the highest median latency at 77.59 ms and the largest spread. This data suggests that Protobuf and MessagePack are more efficient at handling smaller batches in terms of latency. When considering outliers, the analysis reveals additional insights: despite Protobuf's lowest median latency, it has the most outliers, landing at 83,785, indicating many instances where latency spikes significantly. MessagePack exhibits 68,905 outliers, showing that while its median latency is relatively low, it also faces frequent high–latency events, suggesting less stability under certain conditions. Avro, with 16,164 outliers, experiences less frequent high–latency events. While its median latency is higher than that of Protobuf and MessagePack, the fewer outliers suggest relatively more consistent performance. JSON, with only 2341 outliers, indicates occasional spikes in latency. However, JSON's latencies, which are mostly within the whiskers, indicate a higher degree of consistency and predictability compared to other protocols, as it shows fewer extreme deviations.

As the record size increases to 4696 bytes, Protobuf continues to exhibit the lowest latency with the median at 57.41 ms, underscoring its efficiency with larger data sizes. However, MessagePack's latency increases, with the median now at 81.04 ms, almost doubling and surpassing JSON's latency, which slightly improves to 72.60 ms. Avro's latency increased to 78.28 ms, showing variability with the change in record size similar to MessagePack.

The spread increased on all the protocols, being most obvious for Avro and MessagePack, ranging from low latencies around 10 ms to over 200 ms, closely followed by JSON. Including outliers provides a clearer picture of performance variability. Protobuf maintains the lowest median latency but has 8,024 outliers, the highest at around 500 ms, indicating that while generally efficient, it occasionally experiences significant latency spikes. MessagePack, with 4518 outliers, shows less frequent high–latency instances, reflecting more stability than Protobuf. Avro, with 9485 outliers, highlights more frequent high–latency events, the highest one reaching above 600 ms. Yet again, JSON shows the lowest number of outliers, only 586, indicating very few high–latency occurrences, and a high degree of consistency.

At the largest record size of 9312 bytes, Protobuf maintains the lowest median latency at 63.14 ms, demonstrating consistent performance across different record sizes. JSON, Avro, and MessagePack exhibit similar median latencies at this size, 78.09 ms, 75.91 ms, and 80.82 ms respectively, showing relative stability in handling larger batches. MessagePack has the widest spread, followed by JSON and Avro. Including outliers reveals further details: despite the lowest median latency, Protobuf has the highest number of outliers, 14,376, suggesting frequent and significant spikes in latency, the highest ones reaching the 1000 ms mark. MessagePack, with 1,241 outliers, shows some high–latency instances, indicating less stability but better than Protobuf. JSON exhibits 1,517 outliers, showing that while it has a lower median latency than MessagePack, its performance is slightly worse with more outliers. Avro has the lowest number of outliers at 435, indicating variability with frequent high–latency events. This contributes to its overall higher latency and larger spread, yet most of its latencies remain within the whiskers, highlighting its consistency and predictability.

The average latencies for each serialization protocol across different record sizes presented in Table 5, highlight Protobuf's superior performance with the lowest average latencies across all sizes: 46.17 ms for 1176 bytes, 59.19 ms for 4696 bytes, and 67.13 ms for 9312 bytes, despite having the highest number of outliers in total. MessagePack, while competitive at the smallest record size with 51.87 ms, shows significant increases for larger sizes, reaching 84.25 ms for 4696 bytes and 85.99 ms for 9312 bytes, suggesting greater variability in performance as data size grows. Avro exhibits relatively high average latencies: 58.68 ms for 1176 bytes, 83.34 ms for 4696 bytes, and 78.94 ms for 9312 bytes, indicating less consistent performance. JSON has the highest average latency at the smallest (80.21 ms) record size but performs better at the 4696–byte and 9312–byte sizes, with an average latency of 74.29 ms and 80.81 ms respectively, demonstrating variability based on data size.

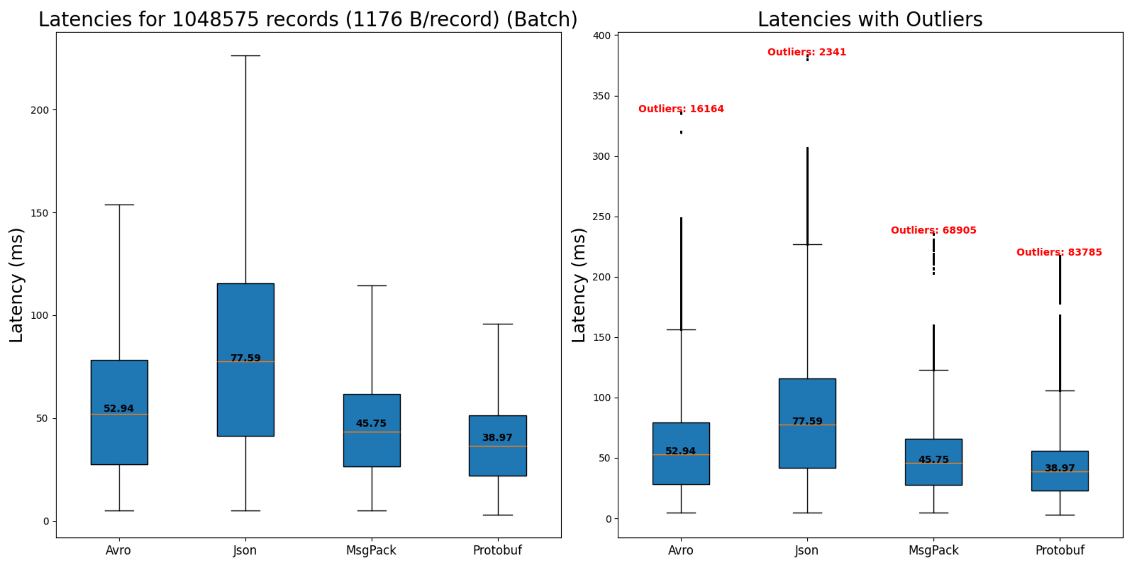

Figure 9: Latency distribution for batch processing 1,048,575 records at 1176 bytes per record. The right plot includes outliers.

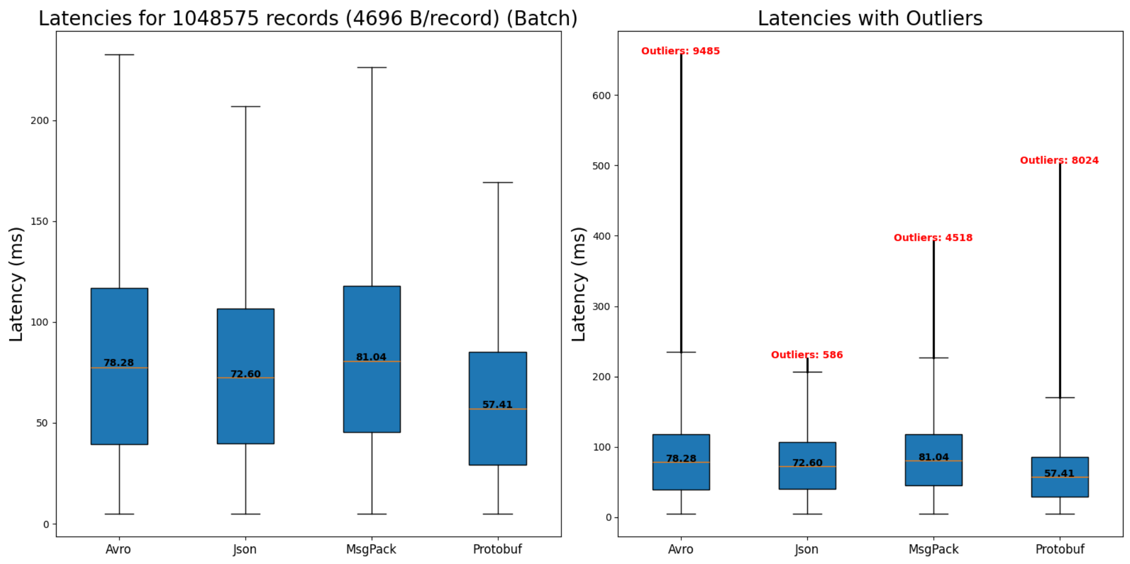

Figure 10: Latency distribution for batch processing 1,048,575 records at 4696 bytes per record. The right plot includes outliers.

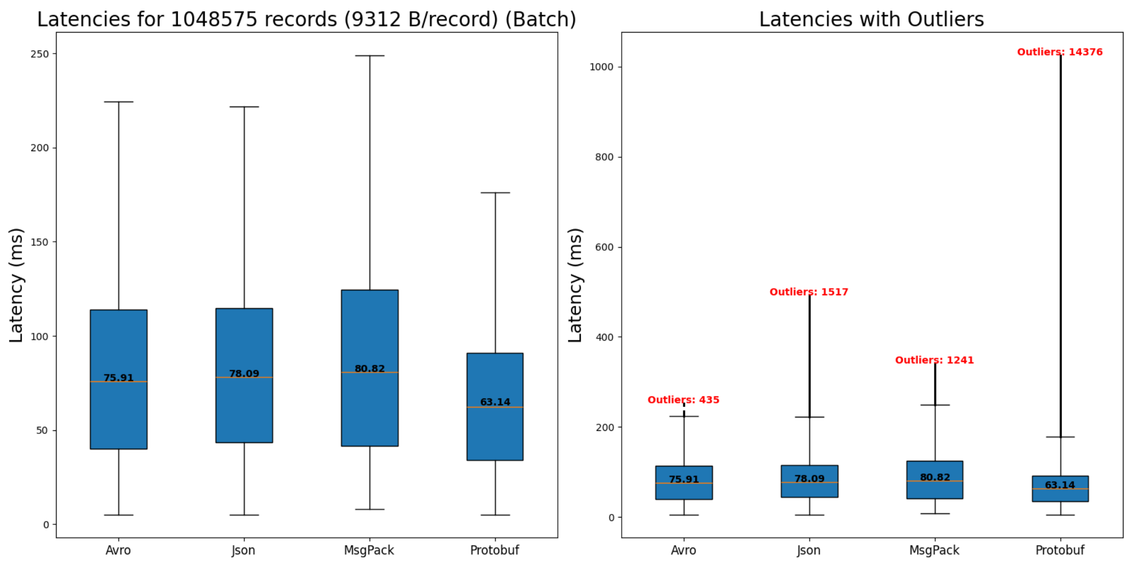

Figure 11: Latency distribution for batch processing 1,048,575 records at 9312 bytes per record. The right plot includes outliers.

Table 4: The average latencies for each protocol and record size for the batch–processing tests.

	Average Latency for Batch–Processing		
Serialization Protocol	**Latency 1176 bytes**	**Latency 4696 bytes**	**Latency 9312 bytes**
`Avro`	58.68 ms	83.34 ms	78.94 ms
`JSON`	80.21 ms	74.29 ms	80.81 ms
`MessagePack`	51.87 ms	84.25 ms	85.99 ms
`Protobuf`	46.17 ms	59.19 ms	67.13 ms

5.2 One by One

Serialization & Deserialization Times

The serialization and deserialization times for the various protocols are aggregated and presented in Figure 12. Across all three record sizes, Protobuf consistently showed the shortest times, with times of 6.979 seconds for 1176 bytes, 29.433 seconds for 4696 bytes, and 52.716 seconds for 9312 bytes, clearly outperforming the other protocols. This was followed by MessagePack, which recorded 9.968 seconds, 40.353 seconds, and 95.281 seconds, respectively, for the same record sizes. Avro and JSON exhibited longer times, with JSON being the least efficient across all sizes.

In a detailed comparison, Protobuf was found to be approximately five times faster than JSON for the record sizes of 1176 bytes (6.979s vs. 35.932s), 4696 bytes (29.433s vs. 158.031s), and 9312 bytes (52.716s vs. 312.013s). MessagePack, the second fastest protocol, consistently performed within a competitive range, never more than twice as slow as Protobuf. For instance, at 9312 bytes, MessagePack's time of 95.281 seconds is roughly 1.81 times that of Protobuf but still considerably faster than Avro and JSON.

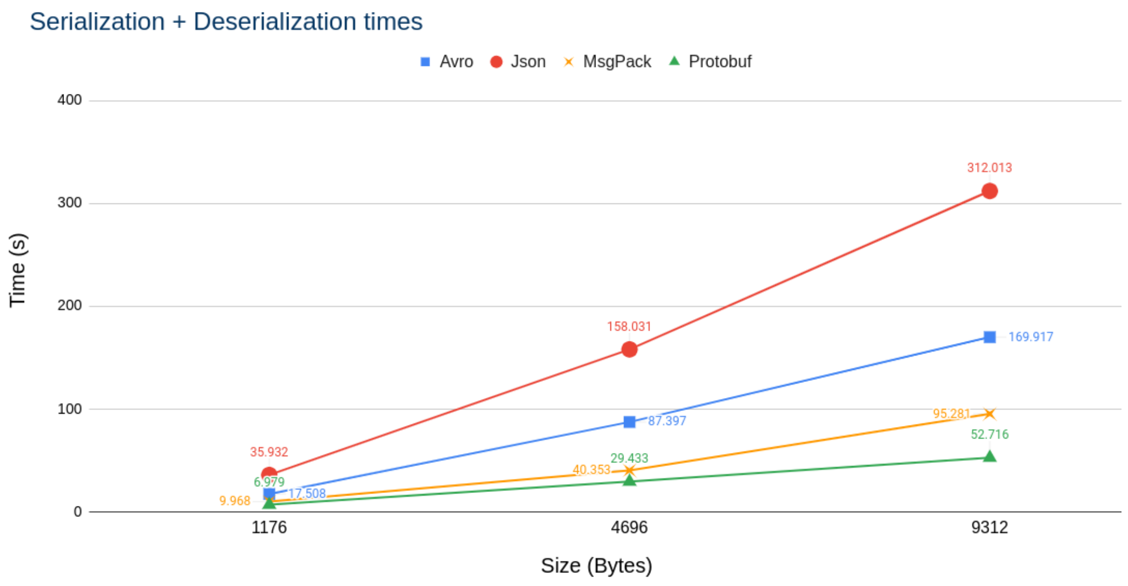

Figure 12: The aggregated serialization and deserialization times for each serialization protocol for the single message tests. The X-axis represents the size of each record, the Y-axis represents the time in seconds, and each line represents a serialization protocol.

Execution Times

The total execution times for the producer and the consumers are aggregated and presented in Figure 13. At the smallest record size, Protobuf performs the best, completing in 28.835 s, slightly outperforming MessagePack at 35.896 s. However, as the record size increases, Protobuf's execution time also increases, allowing MessagePack and Avro to overtake it. MessagePack emerges as the best performer at the two larger sizes completing at 125.583 s for the record size of 4696 bytes, followed by Protobuf and 277.363 s for 9312 bytes, followed by Avro. When comparing the best-performing protocol (MessagePack) with the worst-performing protocol (JSON), MessagePack is consistently at least twice as fast across all record sizes.

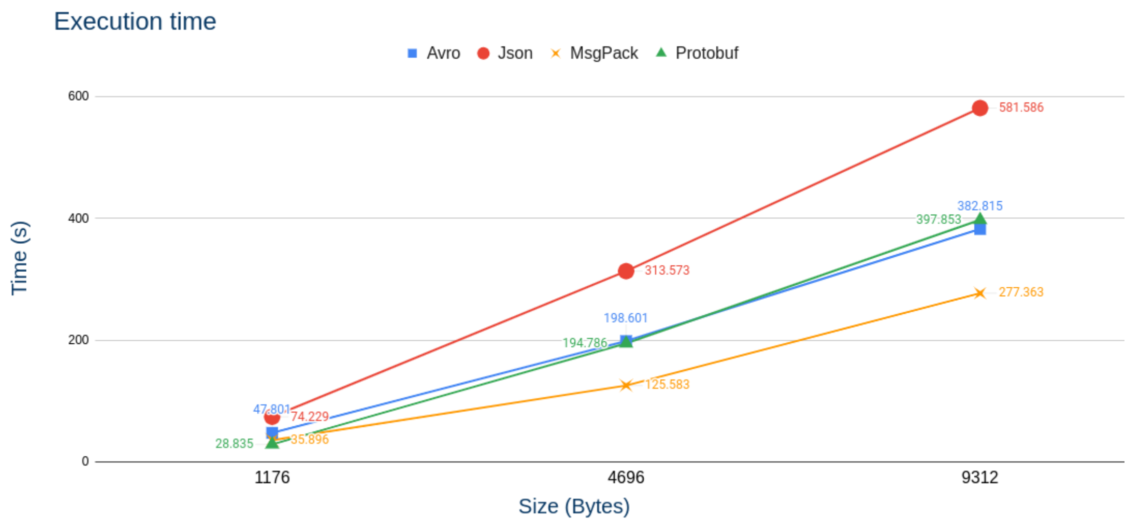

Figure 13: Aggregated producer and consumer times for the single message test for each protocol and each size. The X-axis represents the size of each record, the Y-axis represents the time in seconds, and each line represents a serialization protocol.

Serialization Impact on Total Time

The impact of serialization times on the total execution time varies by protocol and record size, as illustrated in Figure 14. Generally, serialization times increase with the record size across all protocols. Again, Protobuf exhibits the opposite trend, where serialization times decrease as record sizes increase.

For Avro, the proportion of time spent on serialization and deserialization remains relatively consistent across the larger record sizes, with an increase from the smallest record size, accounting for 36.6% at 1176 bytes, increasing slightly to 44% at 4696 bytes, and then to 44.4% at 9312 bytes. JSON shows a slight increase in the serialization and deserialization proportion as record sizes grow, starting from 48.4% at 1176 bytes, increasing to 50.4% at 4696 bytes, and increasing to 54.6% at 9312 bytes. MessagePack also shows an increasing trend, with 27.8% at 1176 bytes, increasing to 32.1% at 4696 bytes, and 34.4% at 9312 bytes. Notably, Protobuf shows the most decrease in serialization and deserialization times as a proportion of total time, starting at 24.2% for 1176 bytes, decreasing to 15.1% at 4696 bytes, and decreasing slightly to 13.3% at 9312 bytes.

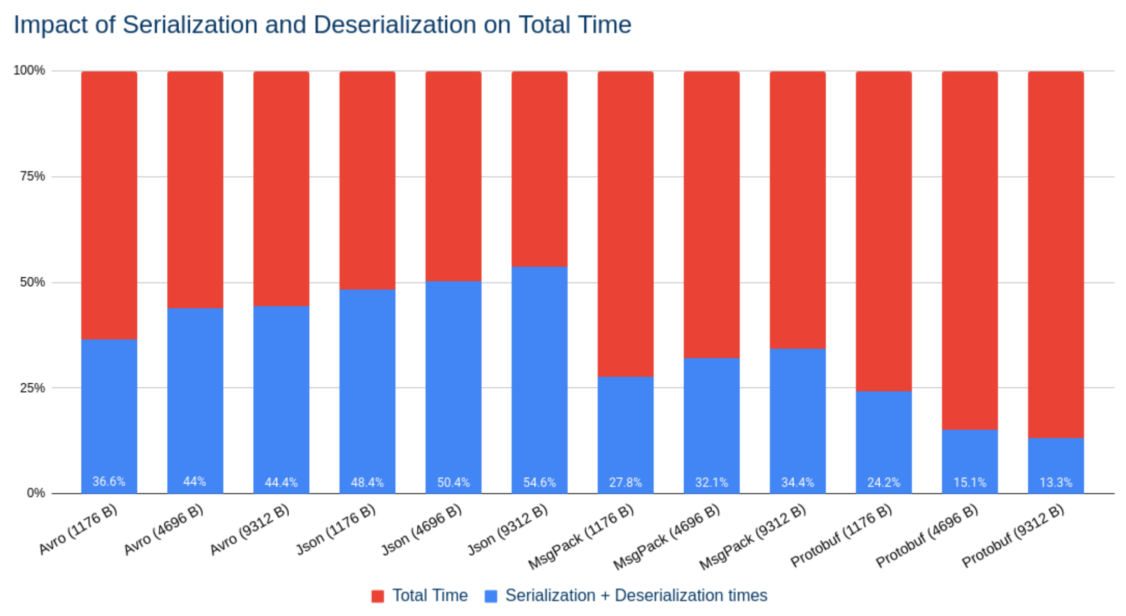

Figure 14: Serialization and deserialization times over the total time for each protocol and size for the single message test. The blue areas represent the percentage of how much of the time was spent serializing or deserializing. The red area represents the Total time, excluding the serialization times.

Throughput

Figure 15 displays the throughput rates for the different serialization protocols. For the smallest record size (1176 bytes), Protobuf achieves the highest throughput at 36,364 records per second, outperforming the second-best protocol, MessagePack at 29,212 records per second. At the record size of 4696 bytes, MessagePack takes the lead with 8,350 records per second, significantly ahead of Protobuf's 5,383 records per second and a little further ahead of Avro's 5,280 records per second. At the largest size of 9312 bytes, MessagePack still leads with 3,781 records per second, which is 38% higher than the closest protocol, Avro at 2,739 records per second. Protobuf shows a throughput of 2,636 records per second. At the same time, JSON performs the worst across all record sizes, with a throughput of 1,803 records per second at 9312 bytes, marking it as at least two times slower than MessagePack across the evaluated record sizes.

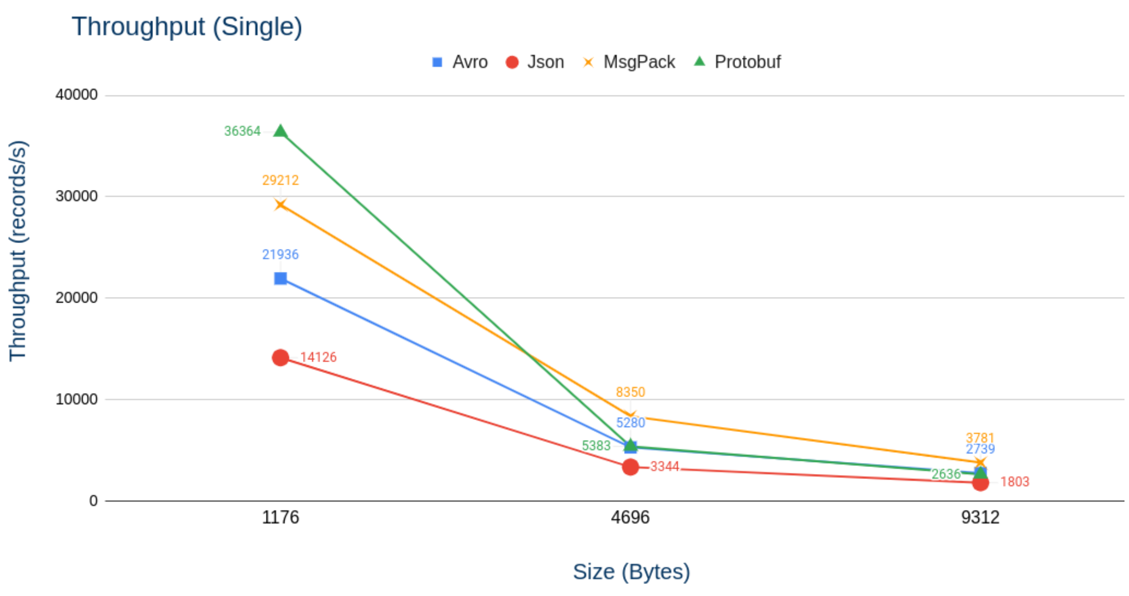

Figure 15: Throughput for the different serialization protocols for the single message–processing tests. The X-axis represents the size of each record, the Y-axis represents the time in seconds, and each line represents a serialization protocol.

Latency

The latency results are presented in three graphs, each representing a record size. The record sizes are 1176 bytes, 4696 bytes, and 9312 bytes, and the latencies for the record sizes are presented in Figures 16, 17 & 18 respectively.

For the smallest data size of 1176 bytes per record, Protobuf demonstrates the best performance with the lowest median latency of 1.68 ms and the lowest spread, indicating highly efficient handling of smaller data sizes. MessagePack maintains a lower spread and has a lower median latency of 2.02 ms compared to Avro's 2.37 ms. JSON has the highest median latency at 7.94 ms as well as the highest spread, suggesting less efficiency in its processing or overhead complications at this record size.

When considering outliers, the analysis provides additional insights. Despite Protobuf's lowest median latency, it has 64,530 outliers, indicating that there are numerous instances where latency spikes significantly. MessagePack exhibits 47,850 outliers, showing that while its median latency is relatively low, it also faces frequent high–latency events, suggesting occasional instability under certain conditions. MessagePack reached the highest latency out of all the protocols for this record size at just over 350 ms. With 14,279 outliers, Avro experiences fewer high–latency events than Protobuf and MessagePack but still has many outliers, contributing to its higher median latency and larger spread. JSON, with 3,004 outliers, has the fewest high–latency events among the protocols. This indicates a significant improvement in consistency and predictability, as the number of outliers is substantially lower than Protobuf's. JSON's latencies, which are mostly within the whiskers, further highlight its relative stability and efficiency despite its higher median latency.

At the 4696 bytes record size, JSON unexpectedly outperforms the other protocols, with a median latency of 3.76 ms and the least spread. Protobuf, performing second best, had the median latency increase to 4.70 ms. Avro and MessagePack had similar median latencies, 5.47 ms and 5.49 ms respectively. Avro's interquartile range spans from 1.8 ms to 6.5 ms, and MessagePack's interquartile range spans from 4.5 ms to 9 ms. This particular result highlights the variability in performance that can occur based on the serialization protocol and the specific characteristics of the data being processed.

Considering outliers, JSON, despite its lowest median latency, has 9,109 outliers, indicating that there are numerous instances where latency spikes significantly. This high number of outliers suggests that while JSON maintains a low median latency, it suffers from frequent high–latency events, impacting its overall consistency. Protobuf, with 8,844 outliers, shows that while it performs well on average, it can experience significant spikes in latency, indicating occasional instability. Avro, with 6,796 outliers, also shows frequent high–latency events, contributing to its higher median latency and larger spread. MessagePack, with 4,141 outliers, has the lowest number of outliers among the protocols, indicating that it experiences fewer latency spikes, but its overall performance is less consistent given its higher spread.

At the 9312–byte record size, Avro showcases the best performance with the lowest median latency of 2.71 ms, effectively handling larger datasets. Protobuf also performs well, presenting a median latency of 3.63 ms. On the other hand, MessagePack records a median latency of 7.37 ms, showing an increase in latency compared to the smaller sizes. JSON, while exhibiting a median latency of 4.18 ms, still outperforms MessagePack but does not match the efficiency of Avro or Protobuf. The interquartile range for MessagePack is the largest among the protocols, indicating a wider variability in its performance. In contrast, JSON maintains a narrower spread, which indicates slightly more consistency than MessagePack but with higher latency than Avro and Protobuf.

When considering outliers, Avro, despite having the lowest median latency, has 20,994 outliers, indicating a significant number of instances where latency spikes considerably. This suggests Avro performs well on average but can experience considerable variability under certain conditions. JSON has 8,588 outliers, reflecting a moderate number of high–latency events. This number of outliers is relatively high compared to Protobuf and MessagePack, indicating less stability in its performance. MessagePack, with 4,359 outliers, shows fewer high–latency events than JSON, but its overall higher median latency and larger spread suggest that it still faces considerable performance variability. Protobuf, with 3,368 outliers, has the fewest outliers among the protocols, indicating relatively stable performance with fewer significant latency spikes.

The average latencies for each serialization protocol across different record sizes, as shown in Table 5, reveal distinct performance characteristics. Avro consistently demonstrates low average latencies, with 5.88 ms for 1176 bytes, 6.11 ms for 4696 bytes, and 5.17 ms for 9312 bytes, indicating stable performance across varying data sizes. JSON shows higher average latencies, particularly for smaller sizes, with 8.06 ms for 1176 bytes and 8.07 ms for 4696 bytes, but it improves to 5.86 ms for 9312 bytes, reflecting less efficiency for smaller data but better performance with larger records. MessagePack exhibits the highest average latency at the smallest size (12.60 ms) but improves to 6.71 ms for 4696 bytes and back to the highest latency at 8.17 ms for 9312 bytes, suggesting that its performance varies significantly with data size. Protobuf, while showing the highest average latency for 1176 bytes at 11.41 ms, performs better with larger sizes, achieving 7.35 ms for 4696 bytes and the lowest average latency of 3.68 ms for 9312 bytes, highlighting its strength in processing larger datasets efficiently.

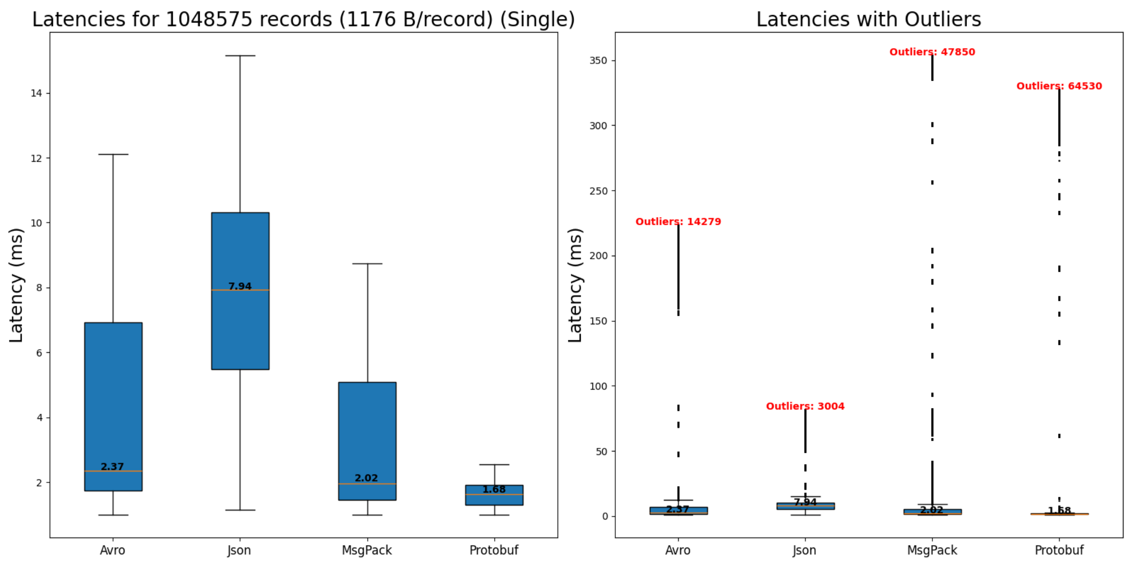

Figure 16: Latency distribution for transmitting 1,048,575 records at 1176 bytes per record, one at a time. The right plot includes outliers.

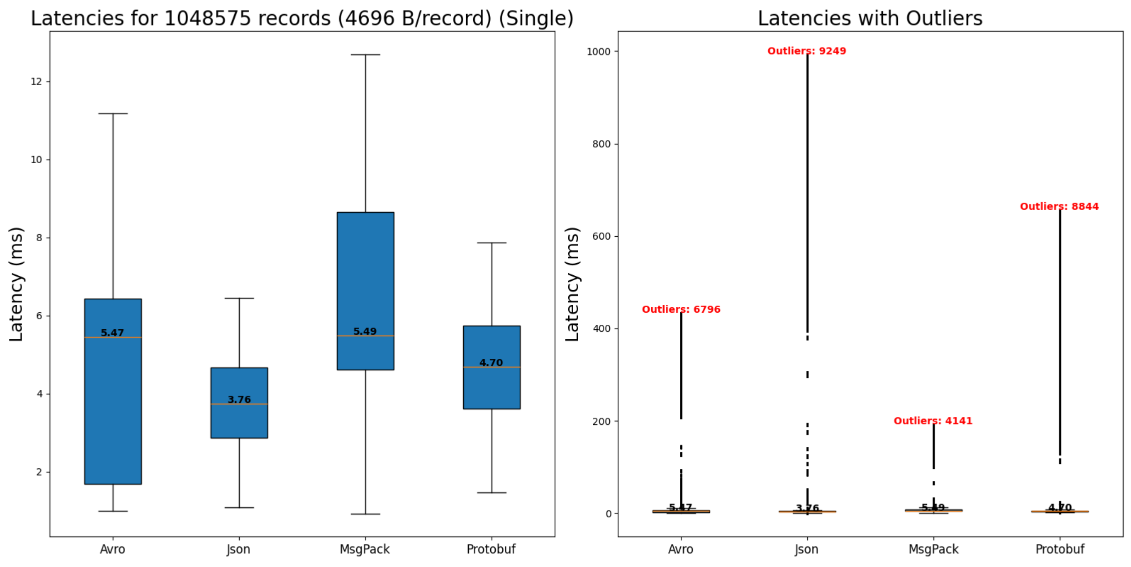

Figure 17: Latency distribution for transmitting 1,048,575 records at 4696 bytes per record, one at a time. The right plot includes outliers.

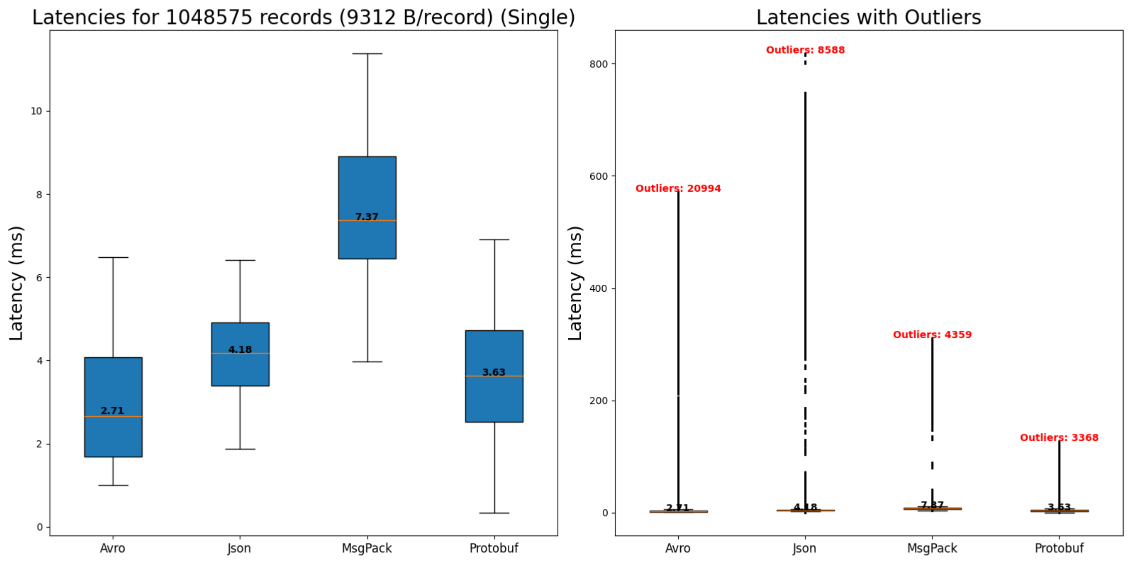

Figure 18: Latency distribution for transmitting 1,048,575 records at 9312 bytes per record, one at a time. The right plot includes outliers.

Table 5: The average latencies for each protocol and record size for the single–message–processing tests.

Average Latency for Single–Message–Processing

Serialization Protocol	Latency 1176 bytes	Latency 4696 bytes	Latency 9312 bytes
`Avro`	5.88 ms	6.11 ms	5.17 ms
`JSON`	8.06 ms	8.07 ms	5.86 ms
`MessagePack`	12.60 ms	6.71 ms	8.17 ms
`Protobuf`	11.41 ms	7.35 ms	3.68 ms

6 Discussion

This section presents a discussion of the results of serialization and deserialization time, execution time, throughput, and latency and the potential causes of these results.

The results of the batch processing and single message tests are similar in the serialization & deserialization and throughput metrics. This behavior is expected because the same amount of serialization and deserialization occurs, and the same amount of data is transmitted in both test cases. However, the tests differ regarding latency because the transmission size is largely increased in the batch processing tests. Therefore, the latency section is split into two sub–sections, each representing the two tests.

Serialization Efficiency & Throughput

From Figures 5 & 12, Protobuf demonstrated superior efficiency in serialization and deserialization times in both batch processing and single message tests across all record sizes. This performance can be attributed to Protobuf's binary format and efficient encoding mechanism, which minimizes size and processing overhead. In contrast, JSON, a text–based format, exhibited the longest serialization times, reflecting its relative verbosity and the higher computational overhead associated with parsing and writing JSON.

The results of the execution time presented in Figures 6 & 13, showed that while Protobuf generally begins with the shortest execution times, it is overtaken by MessagePack as the record size increases. MessagePack's balance between compact binary format and efficient processing appears to scale better with size, making it the preferred choice for larger records under heavy loads. Avro performs similarly to Protobuf, being off by just seconds in both directions for sizes 4696 and 9312 bytes. JSON performed the worst, having twice as long execution times as MessagePack.

One might think that since Protobuf has the quickest serialization and deserialization process and the transmission process is the same for all protocols, it should also have the quickest execution times. Unfortunately, that is not the case due to the inefficiencies in Protobuf's Python library. As explained in Section 4.3, the producer needs to build a data object before serializing it. In the case of Protobuf, the data object is already compiled and included in the Python program (see Section 2.4), but the data object needs values assigned to its attributes. The attributes are set using the function "setattr()", which is included in Protobuf's Python library and seems to be the culprit for Protobufs' success. The data object for the other protocols is built with the built–in data structure "Dictionary" in Python. Hence, the resemblance between the serialization & deserialization results is somewhat proportional to the execution times.

This can be seen when comparing the plots for serialization and deserialization times and execution times (see Figures 5, 6, 12 & 13). The slope for all serialization protocols except for Protobuf, going from the serialization and deserialization times plot to the execution times plot, has the same growth and proportionally the same slope. However, Protobuf's slope on the execution time result has an increased incline which is not proportional to the serialization & deserialization result as the record size increases.

Throughput

The throughput is directly related to the execution time of the tests and is mirrored in Figures 8 & 15. The data suggests that Protobuf has the highest throughput for the lowest size 1176 bytes, and MessagePack has the highest throughput for 4696 and 9312 bytes. There is no clear "winner" in terms of throughput, rather it depends on the use case of the application and the data size to be sent. If an application sends data close to 1176 bytes, the serialization protocol that generates the highest throughput (out of the protocols in question) would be Protobuf. Unfortunately, it is not certain that this result would hold for all record sizes up to 1176 bytes, but it is very likely. If the application sends data larger than 1176 bytes, say around 4696 bytes until 9312 bytes, MessagePack is a better pick in terms of throughput. Again, it is not certain that MessagePack is the best–performing serialization protocol in terms of throughput for sizes larger than 9312 bytes, but it is very likely based on the data.

Batch Processing Latency

In the batch–processing tests, the behavior of latency across different record sizes as shown Table 4, and in Figures 9, 10, & 11 vary very slightly between the different protocols.

Protobuf consistently demonstrates the lowest median latencies across all record sizes, with 38.97 ms for 1176 bytes, 57.41 ms for 4696 bytes, and 63.14 ms for 9312 bytes. This indicates Protobuf's high efficiency in handling batch processing, making it a strong candidate for applications requiring quick and efficient processing of large batches. MessagePack follows with competitive median latencies, particularly at the smallest size (45.75 ms). Avro and JSON show higher median latencies, with JSON exhibiting the highest median latency at the smallest record size (77.59 ms) but performing better at larger sizes.

The presence of outliers significantly affects the overall latency performance. Protobuf, despite its low median latencies, has a substantial number of outliers: 83,785 at 1176 bytes, 8,024 at 4696 bytes, and 14,376 at 9312 bytes. These outliers indicate frequent latency spikes, impacting its predictability and consistency. MessagePack also exhibits a notable number of outliers, with 68,905 at 1176 bytes and 4,518 at 4696 bytes, suggesting less stability under certain conditions. Avro demonstrates relatively more consistent performance with fewer outliers (16,164 at 1176 bytes and 435 at 9,312 bytes). JSON has the fewest outliers across the first two sizes, indicating higher predictability despite its higher median latencies.

The mean latency values, presented in Table 4, confirm that while Protobuf has the lowest average latencies overall, the comparison between median and average latencies highlights its susceptibility to latency spikes. JSON, with fewer outliers, offers more consistent performance, whereas MessagePack and Avro show higher average latencies for 1176 bytes and 4696 bytes, due to the impact of their outliers. These findings underscore the importance of considering both average and median latencies and the outliers' impact when evaluating serialization protocols' performance and stability.

To summarize the discussion, Protobuf consistently demonstrates the lowest median latencies across all record sizes, making it the best performer due to its efficient binary format and quick processing capabilities. For the smallest size of 1176 bytes, MessagePack is the runner–up, showing competitive median latencies but with significant variability due to outliers. JSON unexpectedly performs well for the medium size of 4696 bytes, although it also has many outliers. At the largest size of 9312 bytes, Avro is the runner–up with relatively stable performance despite a higher number of outliers. These findings highlight Protobuf's overall efficiency and the varying reliability of MessagePack, JSON, and Avro depending on the record size.

Single Message Processing Latency

For the single message processing test, Protobuf achieves the lowest median latency of 1.68 ms, but its performance is ruined by many outliers, resulting in an average latency of 11.41 ms. This discrepancy indicates that while Protobuf can deliver excellent performance under optimal conditions, it is prone to substantial latency spikes under less favorable conditions. MessagePack shows similar issues, with a median latency of 2.02 ms but a higher average latency of 12.60 ms due to severe outliers. Avro demonstrates more stable performance with a median latency of 2.37 ms and fewer outliers, leading to a more consistent average latency of 5.88 ms. Despite having the highest median latency of 7.94 ms, JSON has the fewest outliers, resulting in a more predictable average latency of 8.06 ms.

As the record size increases to 4696 bytes, JSON unexpectedly registers the lowest median latency at 3.76 ms and the highest number of outliers, resulting in the highest average latency of 8.07 ms. Protobuf maintains better typical performance with a median latency of 4.70 ms and an average latency of 7.35 ms, although it still experiences variability due to outliers. Avro and MessagePack have similar median latencies, around 5.47 ms. Although MessagePack has fewer outliers, JSON's lower outlier values lead to a closer alignment between its median and average latencies, showing slightly better stability.

At the largest record size of 9312 bytes, Avro achieves the lowest median latency of 2.71 ms but is impacted by many outliers, resulting in an average latency of 5.17 ms. Protobuf shows strong performance with a median latency of 3.63 ms and the fewest outliers, maintaining a stable average latency of 3.68 ms. MessagePack, with a median latency of 7.37 ms and a noticeable number of outliers, has an average latency of 8.17 ms, indicating a minor variability. JSON, with a median latency of 4.18 ms and more outliers than Protobuf, achieves an average latency of 5.86 ms, reflecting slightly worse stability than MessagePack and even less so than Protobuf.

To summarize the discussion, Protobuf achieves the lowest median latency of 1.68 ms for the smallest size (1176 bytes) but is significantly impacted by a high number of outliers, resulting in a higher average latency of 11.41 ms. MessagePack is the runner-up with a median latency of 2.02 ms, though it also suffers from severe outliers, leading to an average latency of 12.60 ms. For the medium size (4696 bytes), JSON unexpectedly registers the lowest median latency at 3.76 ms but has the highest number of outliers, resulting in an average latency of 8.07 ms. Protobuf maintains better typical performance for this size, with a median latency of 4.70 ms and a more stable average latency of 7.35 ms. At the largest size (9312 bytes), Avro achieves the lowest median latency of 2.71 ms but is affected by many outliers, leading to an average latency of 5.17 ms. Protobuf again shows strong performance with a median latency of 3.63 ms and the fewest outliers, maintaining a stable average latency of 3.68 ms, making it the most consistent performer for larger sizes.

Potential Latency Causes

Since the tests are conducted in the public cloud, results are inconsistent from run to run. The results depend on the time of day the tests were performed, and the resource usage of tenants, such as network bandwidth and hardware resources. The outliers presented in the latency figures are anomalies for the specific runs, meaning they deviate from the majority of the latencies. The anomalies can be a small fraction of the total number of latencies, or be as high as 6.15 % (in the case of Protobuf at 1176 bytes).

The outliers could also result from a client bottleneck, where the producer produces messages faster than the consumers can consume them, leading to messages spending more time in transit. Another possibility for the outliers is the implementation of Kafka. Three partitions were assigned to each Kafka topic, allowing for 3 simultaneous consumers to consume messages from the topic. The producer was configured to distribute the messages in a “Round–Robin” fashion (equally) across all partitions, but in reality, it sometimes favored one partition more than the others, which resulted in longer waiting times for those messages.

Summary

To summarize the discussion and to answer the research questions posed in the introduction (see Section 1.1), the first question (RQ1) can be answered by a recap of the discussion.

When it comes to throughput, Protobuf has the highest throughput for the smallest size (1176 bytes), while MessagePack excels for larger record sizes (4696 and 9312 bytes). In the batch–processing test, MessagePack’s throughput, 34,254 records per second, was not too far off from Protobuf’s 36,945 records per second, meaning it is a strong candidate. Avro had an average performance, and JSON had the worst performance for all sizes, being at least twice as slow as MessagePack.

Regarding latency, starting with the batch–processing tests, Protobuf consistently demonstrates the lowest median and average latencies across all record sizes (1176, 4696, and 9312 bytes), making it highly efficient for batch processing. However, Protobuf’s performance is affected by significant outliers, leading to occasional latency spikes. MessagePack follows with competitive median latencies, particularly at smaller sizes, but also experiences variability due to outliers. Avro and JSON show higher median latencies, with JSON performing the worst at smaller sizes but improving at larger sizes. JSON, however, has the fewest outliers, ensuring more predictable performance.

For the single message tests, Protobuf again achieves the lowest median latency for the smallest size (1176 bytes) but is significantly impacted by outliers, resulting in a higher average latency. For the medium and large sizes, Protobuf maintains better performance with fewer outliers and stable median latencies. JSON unexpectedly registers the lowest median latency for the medium size (4696 bytes) but suffers from many outliers, impacting its average latency. Avro achieves the lowest median latency for the largest size (9312 bytes) at 2.71 ms but is affected by outliers, and Protobuf achieves the lowest average latency.

This summary can answer the second research question (RQ2). Comparing the results for the different tests, and witnessing that JSON performed the worst out of all the protocols in terms of throughput, serialization & deserialization, and execution time, it is safe to say that it is worth migrating from JSON to any other serialization protocol to gain benefits. Migrating to MessagePack would yield at least two times more throughput and finish executing two times faster. This is also true if one migrates to Protobuf, but only for the lowest size of 1176 bytes. Avro gains only a slight advantage in terms of throughput and execution time over JSON and is not nearly as good of a choice as MessagePack.

Regarding latency, specifically batch–processing, Protobuf has a two times lower median latency than JSON, at 38.97 ms compared to 77.59 ms at the lowest size (1176 bytes). It also has a lower median latency for the other sizes, but not as significant. Protobuf also has a lot of outliers, which suggests variability and uncertainty. It would almost not be worth migrating to Protobuf solely on batch–process latency results because they are not significant enough and would barely be noticeable. As for Avro and MessagePack, it is not worth migrating to, based solely on batch–process latency results.

Finally, when looking at the single message latency, all the protocols perform better than JSON for the smallest size, showing a 3-4 times better median latency at the cost of outliers and variability. For the other two sizes, only Avro can really compare, showcasing slightly lower latencies, but barely noticeable, and again at the cost of outliers and variability. It is only worth migrating to Avro from JSON if the slightest latency improvement benefits the application's use case, with a trade–off of potential anomalies. Basing the choice solely on the latency results is not a good idea.

7 Conclusion & Future Work

This section summarizes conclusions from the results and future work to improve the tests.

7.1 Conclusion

This thesis investigates the performance of different serialization protocols in Apache Kafka, mainly focusing on their impact on throughput and latency across varying data sizes. It also touches on serialization & deserialization times, and execution times. The serialization protocols evaluated are Protocol Buffers (Protobuf), MessagePack, Apache Avro, and JSON.

The tests were conducted in a public cloud environment using Apache Kafka. Each protocol was evaluated under batch processing and single message scenarios across three different record sizes: 1176 bytes, 4696 bytes, and 9312 bytes. Metrics such as serialization and deserialization times, execution times, throughput, and latency were measured. The latency was further analyzed by considering both median latencies and the presence of outliers, which significantly impact overall performance.

In summary, the results indicate that migrating from JSON to any other protocol would increase the throughput, mainly MessagePack or Protobuf (for the smallest size), which yields substantial performance improvements in Apache Kafka. For batch processing, Protobuf generally provides the lowest median latency but is affected by significant outliers. However, migrating to Protobuf solely on batch—processing latency results is not worth it because the difference in latencies is negligible. In single–message processing, it is worth migrating to Avro or Protobuf if the slightest improvement in latency benefits the use case of the application, with a trade–off of potential anomalies.

7.2 Future Work

While the study provides comprehensive insights into the performance of each protocol under various conditions, it does not account for network behavior, which can influence performance outcomes. Future work could be done by isolating the network to prevent external interferences, such as bandwidth consumption by other tenants in the public cloud, as well as VMs in different racks across the data center, which could ensure more accurate and reliable measurements. This controlled environment would mitigate external network interferences, offering a clearer understanding of the serialization protocols' performance in an ideal setting.

Moreover, while this study provides significant insights into the performance impacts of various serialization protocols in Apache Kafka, future research could expand on these findings by exploring additional configurations and settings within Kafka. Specifically, future work could investigate the effects of different compression algorithms on performance, as different algorithms may influence the speed and efficiency of data transmission. Additionally, examining the impact of varying partition counts could provide a deeper understanding of Kafka's scalability. The performance implications of different producer and consumer counts should also be considered, as these factors can significantly affect throughput and latency. Another important area for future research is the effect of enabling encryption within Kafka. Encryption can add overhead, impacting the overall performance, but securing the contents of the data that is being transmitted is in the interest of companies. These explorations could provide a more holistic view of Kafka's performance in diverse real–world scenarios and contribute to the optimization of Kafka–based systems.

Additionally, exploring the impact of newer or less common serialization frameworks could also yield beneficial insights, particularly as the field of data serialization continues to evolve rapidly with advancements in technology and methodology.

References

[1] Amazon. *Throughput vs latency - difference between computer network*. https://aws.amazon.com/compare/the-difference-between-throughput-and-latency/. Accessed: 22 February 2024.

[2] Amazon. *What is latency? - latency explained - AWS*. https://aws.amazon.com/what-is/latency/. Accessed: 26 February 2024.

[3] *Apache Avro Python library fastavro*. https://github.com/fastavro/fastavro. Accessed: 13 May 2024.

[4] Apache Software Foundation. *Apache Avro*. https://avro.apache.org/docs/. Accessed: 13 February 2024.

[5] Apache Software Foundation. *Apache Kafka*. https://kafka.apache.org/. Accessed: 12 February 2024.

[6] Berker Agir. *Benchmarking Kafka producer throughput with Quarkus*. https://developers.redhat.com/articles/2021/07/19/benchmarking-kafka-producer-throughput-quarkus#analyzing_producer_message_throughput. Accessed: 8 May 2024.

[7] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259.

[8] David Carrera Castillo, Jonathan Rosales, and Gustavo A Torres Blanco. "Optimizing binary serialization with an independent data definition format". In: *International Journal of Computer Applications* 180.28 (2018), pp. 15–18.

[9] Chandrasekar Ganapathy. *Azure Kubernetes Service Multitenancy*. https://community.hitachivantara.com/blogs/chandrasekar-ganapathy/2023/05/17/azure-aks-multitenancy-primer. Accessed: 8 May 2024.

[10] Cloudflare. *What is latency? | how to fix latency | cloudflare*. https://www.cloudflare.com/learning/performance/glossary/what-is-latency/. Accessed: 26 February 2024.

[11] Confluent. *How to optimize your Kafka producer for throughput*. https://developer.confluent.io/tutorials/optimize-producer-throughput/confluent.html. Accessed: 10 May 2024.

[12] Philippe Dobbelaere and Kyumars Sheykh Esmaili. "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*. 2017, pp. 227–238.

[13] Malin Eriksson and Victor Hallberg. "Comparison between JSON and YAML for data serialization". In: *The School of Computer Science and Engineering Royal Institute of Technology* (2011), pp. 1–25.

[14] Guo Fu, Yanfeng Zhang, and Ge Yu. "A fair comparison of message queuing systems". In: *IEEE Access* 9 (2020), pp. 421–432.

[15] Geoffrey Hunter. *A comparison of serialization formats*. https://blog.mbedded.ninja/programming/serialization-formats/a-comparison-of-serialization-formats/. Accessed: 22 February 2024. 2019.

[16] Google LLC. *Protocol Buffers*. https://protobuf.dev/overview/. Accessed: 13 February 2024.

[17] Jean Carlo Hamerski et al. “Evaluating serialization for a publish-subscribe based middleware for MPSoCs”. In: *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE. 2018, pp. 773–776.

[18] Geoff Huston. “Protocol basics: the network time protocol”. In: *APNIC Lab* 10.03 (2014).

[19] IBM. *Throughput*. https://www.ibm.com/docs/en/informix-servers/12.10?topic=performance-throughput. Accessed: 22 February 2024.

[20] *JSON*. https://www.json.org/json-en.html. Accessed: 13 February 2024.

[21] Kałuża, S. *Data serialization tools comparison: Avro vs Protobuf, SoftwareMill.* https://softwaremill.com/data-serialization-tools-comparison-avro-vs-protobuf/. Accessed: 22 February 2024. 2023.

[22] Kanawat, A.S. *JSON vs Messagepack: The Battle of Data Efficiency, Choosing Between JSON and MessagePack: Pros, Cons, and Performance Comparison.* https://www.linkedin.com/pulse/json-vs-messagepack-battle-data-efficiency-akshay-singh-kanawat-7dx2c/. Accessed: 22 February 2024.

[23] Khalfe, A. *Data serialization formats: AVRO, Protocol Buffers, and JSON, The Talent500 Blog.* https://talent500.co/blog/data-serialization-formats-avro-protocol-buffers-and-json/. Accessed: 22 February 2024. 2023.

[24] Paul Le Noac’h, Alexandru Costan, and Luc Bougé. “A performance evaluation of Apache Kafka in support of big data streaming applications”. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 4803–4806.

[25] Kazuaki Maeda. “Performance evaluation of object serialization libraries in XML, JSON and binary formats”. In: *2012 Second International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*. IEEE. 2012, pp. 177–182.

[26] Andrew S. Marteen Van Steen. “Distributed systems Principles and Paradigms”. In: 2017.

[27] Messagepack. *MessagePack: It’s like JSON. but fast and small.* https://msgpack.org/index.html. Accessed: 21 February 2024.

[28] *MessagePack for Python*. https://github.com/msgpack/msgpack-python. Accessed: 13 May 2024.

[29] Microsoft. *Azure Kubernetes Service (AKS)*. https://azure.microsoft.com/en-us/products/kubernetes-service. Accessed: 7 May 2024.

[30] Microsoft. *Fsv2-series*. https://learn.microsoft.com/en-us/azure/virtual-machines/fsv2-series. Accessed: 10 May 2024.

[31] Misja.com. *Epoch & Unix Timestamp Conversion Tools*. https://www.epochconverter.com/. Accessed: 10 May 2024.

[32] Elias Norgren. *How compression affects the use of message queues*. 2022.

[33] Obkio. *How to measure latency*. https://obkio.com/blog/how-to-measure-latency/. Accessed: 26 February 2024.

[34] Bo Petersen et al. “Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the internet of things”. In: *2017 Computing Conference*. IEEE. 2017, pp. 1339–1346.

[35] Ida Mengyi Pu. *Fundamental data compression*. Butterworth-Heinemann, 2005.

[36] Python Software Foundation. *JSON encoder and decoder*. https://docs.python.org/3/library/json.html. Accessed: 13 May 2024.

[37] Rabbitmq. *RabbitMQ: Easy to use, flexible messaging and streaming*. https://www.rabbitmq.com/. Accessed: 1 March 2024.

[38] Amir Rabiee. *Analyzing Parameter Sets For Apache Kafka and RabbitMQ On A Cloud Platform*. 2018.

[39] Shantanu Deshmukh. *Message compression in Apache Kafka*. https://developer.ibm.com/articles/benefits-compression-kafka-messaging/. Accessed: 1 March 2024.

[40] Gwen Shapira et al. *Kafka: the definitive guide*. " O'Reilly Media, Inc.", 2021.

[41] Keshav Sud, Pakize Erdogmus, and Seifedine Kadry. *Introduction to Data Science and Machine Learning*. BoD–Books on Demand, 2020.

[42] Audie Sumaray and S Kami Makki. "A comparison of data serialization formats for optimal efficiency on a mobile platform". In: *Proceedings of the 6th international conference on ubiquitous information management and communication*. 2012, pp. 1–6.

[43] Saurabh Zunke and Veronica D'Souza. "Json vs xml: A comparative performance analysis of data exchange formats". In: *IJCSN International Journal of Computer Science and Network* 3.4 (2014), pp. 257–61.

A Serialization Formats

JSON

```
{
  "first_name": "Alice",
  "last_name": "Andreson",
  "age": 30,
  "phone_number": "123-456-7890"
}
```

MessagePack

```
\x84\xaafirst_name\xa5Alice\xa9last_name\xa8Andreson\xa3age\x1e\xac
phone_number\xac123-456-7890
```

Protocol Buffers

```
\n\x05Alice\x12\x08Andreson\x18\x1e"\x0c123-456-7890
```

Avro

```
\nAlice\x10Andreson<\x18123-456-7890
```