G-Tech : Jurnal Teknologi Terapan
Vol. 10, No. 3, July 2026, p. 1117-1130
E-ISSN: 2623-064X | P-ISSN: 2580-8737

Comparison of JSON and MessagePack Serialization Performance on End-to-end latency over HTTP Protocol

Muhammad Ilham Dzurinadib[1]✉, Muhamat Maariful Huda[2], Tito Prabowo[3]

[1, 2, 3] Ilmu Komputer, Fakultas Ilmu Eksakta, Universitas Nahdlatul Ulama Blitar, Indonesia

✉ **Corresponding Author :** Muhammad Ilham Dzurinadib (e-mail: dzrnadb@gmail.com)

Article Information

Article History

Received : May 25, 2026
Revised : June 17, 2026
Published : July 10, 2026

***Keywords*:**

Data Serialization;
JSON;
MessagePack;
HTTP Communication;
End-to-end latency.

ABSTRACT

The increasing use of HTTP-based communication in modern service-oriented systems requires efficient data serialization formats to reduce communication latency and transmission overhead. This study compares the performance of JSON and MessagePack serialization formats in HTTP communication using the Go programming language. Performance was evaluated using synthetic datasets containing 10, 100, 1000, and 5000 records. The measured parameters included payload size, encoding latency, decoding latency, and End-to-end latency. The results show that MessagePack consistently outperformed JSON across all evaluated metrics. For the largest dataset containing 5000 records, MessagePack reduced End-to-end latency from 13.269 ms to 3.894 ms and generated a smaller payload size (335.87 KB) than JSON (350.53 KB). The performance advantage became more pronounced as the dataset size increased, particularly in decoding and End-to-end latency processes. These findings indicate that MessagePack is an efficient alternative to JSON for HTTP-based communication systems requiring low latency, bandwidth efficiency, and high-performance data exchange.

INTRODUCTION

Recent studies have highlighted the increasing demand for efficient data exchange in service-oriented architecture (SOA) and application programming interface (API) systems as modern applications become more distributed and service-oriented (Zahra & Ardiansyah, 2026) . Efficient communication mechanisms are essential because data exchange performance directly affects the responsiveness and scalability of web services (Bermbach & Wittern, 2020). One of the most widely used data exchange formats is JavaScript Object Notation (JSON). JSON is widely adopted because it provides a simple and lightweight mechanism for representing structured data, making it suitable for data exchange across different platforms and applications (Keiser, 2024).Its widespread use in web services and API development has contributed to its role as a standard communication format in modern distributed systems (Sutisnawati et al., 2025). Consequently, JSON is commonly used in HTTP-based communication, RESTful APIs, and microservice architectures (Yudanto et al., 2025).

Although JSON offers excellent interoperability and ease of use, its text-based representation tends to generate larger payload sizes than binary serialization formats (Viotti & Kinderkhedia, 2022). Furthermore, parsing and processing JSON documents introduce additional computational overhead, particularly when handling large volumes of structured data (Keiser, 2024). Previous studies have also shown that improving JSON parsing performance requires specialized optimization techniques, indicating the computational complexity of text-based data

processing (Langdale & Lemire, 2019). These limitations may increase communication latency and reduce overall application performance in bandwidth-sensitive environments (Huseynov et al., 2024). As a result, binary serialization formats have gained increasing attention because they generally provide more compact payload representations and lower communication overhead than text-based formats, making them suitable for bandwidth-sensitive and latency-critical applications (Viotti & Kinderkhedia, 2022).

As an alternative to text-based serialization formats, binary serialization technologies such as MessagePack are designed to represent structured data in a more compact form. Compact binary representations can reduce payload size and improve data transmission efficiency, particularly in communication environments with bandwidth constraints (Araque et al., 2022; Huseynov et al., 2024). Previous studies have also demonstrated that serialization and deserialization performance significantly affect overall communication efficiency and application responsiveness, highlighting the importance of selecting an appropriate serialization mechanism (Marcelino et al., 2025). Consequently, binary serialization formats are considered well suited for service communication scenarios that require low latency (Valiveti, 2025). Their compact representation can also improve resource utilization and communication efficiency by reducing data transmission overhead (Araque et al., 2023).

Several previous studies have investigated the performance of serialization formats in distributed systems and service-based communication. (Viotti & Kinderkhedia, 2022) benchmarked multiple JSON-compatible serialization formats and reported that binary-based formats generally produce smaller payload sizes than JSON. Similarly, (Maltsev & Amin, 2024) demonstrated that serialization format selection significantly influences inter-service latency in distributed environments. (Cummings, 2024) further showed that binary serialization technologies such as Protocol Buffers and FlatBuffers can reduce serialization overhead and improve processing efficiency compared to text-based formats (Balalayeva et al., 2024). Collectively, these studies suggest that the benefits of binary serialization formats extend beyond payload reduction, influencing both processing efficiency and communication latency.

Despite these findings, several research gaps remain. Previous studies have reported that serialization and deserialization may account for a substantial portion of total communication overhead in distributed workflows, motivating further investigation under realistic HTTP-based communication scenarios (Lu et al., 2024). In addition, many experiments were conducted using relatively small datasets or different programming language ecosystems, making it difficult to assess how serialization formats behave in large-scale HTTP services implemented in Go (Myastovskiy, 2024). Consequently, further investigation is needed to evaluate the impact of serialization format selection on payload size, encoding latency, decoding latency, and End-to-end latency performance under realistic HTTP-based communication scenarios (Proos, 2020).

Based on the aforementioned issues, this research was conducted to compare the performance of JSON and MessagePack serialization formats at the application level using the Go programming language. The study focuses on measuring payload size, encoding latency and decoding latency, and HTTP-based End-to-end latency using data size variations of 10, 100, 1000, and 5000 records. This study extends existing research by providing a comprehensive evaluation of JSON and MessagePack in an HTTP-based End-to-end latency environment using the Go programming language, while simultaneously analyzing payload size, encoding latency, decoding latency, and End-to-end latency across multiple dataset scales.

Unlike previous studies that primarily focused on isolated serialization benchmarks or relatively small datasets, this study evaluates JSON and MessagePack within a complete HTTP-based End-to-end latency workflow implemented in Go. Furthermore, the experiments employ dataset sizes of up to 5000 records, enabling the assessment of serialization performance under larger-scale communication scenarios.

The main contributions of this research include comprehensive performance measurements of JSON and MessagePack serialization using four variations of data sizes; implementation of testing using the Go programming language, which has high runtime efficiency

and concurrency support; measurement of End-to-end latency HTTP-based latency covering the encoding process, data transmission, and response decoding; as well as the development of a controlled testing methodology and synthetic dataset so that the experiments can be reproduced in future research.

RESEARCH METHOD

This study employed an experimental benchmarking approach to compare the performance of JSON and MessagePack serialization formats in HTTP-based communication using the Go programming language. The research methodology was designed to evaluate the impact of serialization format selection on payload size, encoding latency, decoding latency, and End-to-end latency under controlled testing conditions. The experimental procedure consists of several stages, including testing environment preparation, system architecture design, synthetic dataset generation, payload and latency measurement, and statistical analysis of the collected results. By using a controlled and reproducible testing framework, the study aims to provide a fair comparison between the two serialization formats and to ensure the reliability of the obtained performance measurements.

Testing Environment Specifications

The testing in this research was conducted using specific hardware and software to ensure that the benchmarking process runs consistently and in a controlled manner, achieving the desired results. The specifications of the testing environment used can be seen in Table 1.

Table 1.Testing Environment Specifications

Komponen	Spesifikasi
Operating system	Microsoft Windows 10 Pro 64-bit
Processor	Intel Core i5-6300U CPU @ 2.40 GHz
RAM	8 GB DDR4
Architecture	x64-based PC
Language	Go 1.21.5
JSON Library	encoding/json (standard library)
MessagePack	github.com/vmihailenco/msgpack/v5
HTTP Server	net/http (standard library)
HTTP Method	HTTP POST
Protocol	HTTP/1.1
Test Iteration	1000 iterations
Warm-up	200 iterations
Network Topology	Localhost (127.0.0.1)

The specifications of the testing environment are used to ensure that the benchmarking process is conducted under consistent system conditions, so that the measurement results of serialization performance can be obtained in a more stable and representative manner (Pargaonkar, 2023).

The Go programming language was chosen because it has high runtime efficiency, concurrency support through goroutines, and a lightweight and stable standard HTTP library for implementing network-based services (Yuan et al., 2020; Valiveti, 2025) . In addition, Go is widely used in the development of microservices and modern backend systems, making it relevant for testing the performance of HTTP-based data communication (Ardiansyah & Fatwanto, 2022). A warm-up phase of 200 iterations and a main measurement phase of 1000 iterations were used to

obtain more stable, consistent, and representative benchmarking results (Viotti & Kinderkhedia, 2022).

System Testing Architecture

The testing architecture in this study uses an HTTP-based client-server model as shown in Figure 1. The client generates synthetic test data using the makePayload(n) function with dataset sizes of 10, 100, 1000, and 5000 records. The data is then serialized using JSON and MessagePack to produce a payload that is sent via HTTP POST to the /echo endpoint on the server. The server receives the request and returns the payload without performing any additional processing. After the response is received, the client performs a deserialization process to convert the payload back into an object.

Encoding latency is measured before the payload is sent via HTTP POST, while decoding latency is measured after the response is received and deserialization is performed on the client side. End-to-end latency is calculated from the start of the serialization process until the deserialization of the response is completed.

The parameters analyzed include encoding latency, decoding latency, payload size, and End-to-end latency, which encompasses the serialization process, HTTP communication, and response deserialization (Marcelino et al., 2025). The test results were then analyzed using statistical parameters such as average (avg), median (p50), and the 95th percentile (p95).

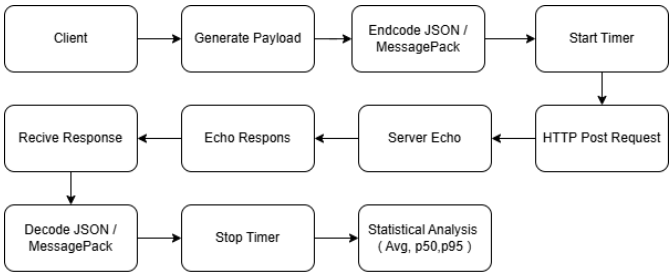

Figure 1. System Testing Architecture

Figure 1 illustrates the experimental workflow used to evaluate serialization performance in an end-to-end HTTP communication scenario. The architecture was designed to isolate the impact of serialization format selection by minimizing additional server-side processing and using a controlled localhost environment. Through this approach, the measured latency primarily reflects the effects of payload generation, serialization, network transmission, and deserialization processes.

HTTP communication scenario. The architecture was designed to isolate the impact of serialization format selection by minimizing additional server-side processing and using a controlled localhost environment. Through this approach, the measured latency primarily reflects the effects of payload generation, serialization, network transmission, and deserialization processes. This design ensures a fair comparison between JSON and MessagePack and improves the reproducibility of the benchmarking results.

Design and Preparation of Test Data

The test data in this study is generated using the makePayload(n) function, which produces objects of the Payload type. The parameter n represents the amount of data used, allowing for performance testing of serialization at various load scales.

The payload structure consists of the attributes id, name, tags, scores, meta, and note. The tags and scores attributes are generated dynamically using loops n times, resulting in a variable payload size. The data structure is designed to represent the common data exchange patterns used in API-based services and modern HTTP communication.

Table 2 summarizes the structure of the payload used throughout the benchmarking process. The payload consists of six fields representing commonly exchanged data types in HTTP-based applications, including primitive values, arrays, and key-value metadata. This combination of data types was intentionally selected to simulate realistic API communication scenarios and to provide a representative workload for evaluating serialization performance.

Table 2. Structure of the Benchmark Payload

Field	Type	Description
ID	int	Unique identifier
Name	string	Benchmark object name
Tags	[]string	Dynamic list of tags
Scores	[]int	Dynamic numerical values
Meta	Map [string] string	Additional metadata
Note	string	Benchmark description

The payload structure combines scalar values, arrays, and nested key-value data to reflect the heterogeneous nature of information commonly exchanged in modern web services. In particular, the dynamically generated tags and scores fields contribute directly to payload growth as the dataset size increases, allowing the benchmark to evaluate how serialization performance is affected by increasing data volume and complexity.

To further illustrate how the same payload is represented by different serialization formats, examples of JSON and MessagePack representations are presented below.

Example JSON representation:

{
"id": 1,
"name": "User Bench",
"tags": ["tag-0", "tag-1", "tag-2"],
"scores": [0, 7, 14],
"meta": {
"app": "bench",
"proto": "http"
},
"note": "simple benchmark payload"
}

Example MessagePack representation (hexadecimal view):

86 A2 69 64 01 A4 6E 61 6D 65 AA 55 73 65 72 20 42 65 6E 63 68 ...

Although both representations contain identical information, JSON stores data in a human-readable text format using field names, quotation marks, and structural delimiters. In contrast, MessagePack encodes the same information into a compact binary representation. Consequently, MessagePack generally requires fewer bytes to represent the same payload, reducing transmission overhead and improving communication efficiency during HTTP-based data exchange (Cummings, 2024).

Variations in data sizes of 10, 100, 1000, and 5000 records are used to represent different data exchange scenarios, ranging from small payloads to large-scale payloads. These dataset sizes were selected to evaluate how serialization performance changes as payload complexity and volume increase. By testing multiple workload levels, the study can observe the scalability characteristics of JSON and MessagePack and identify performance differences that may become more pronounced under larger data processing conditions.

Implementation of Payload Size and Latency Measurement

The measurements were conducted using data size variations of 10, 100, 1000, and 5000 records. Before the main measurements were carried out, the system performed a warm-up phase of 200 iterations to reduce the impact of initial initialization processes such as cache warming, garbage collection stabilization, and HTTP connection formation (Viotti & Kinderkhedia, 2022).

The main measurement was conducted for 1000 iterations to enhance the statistical validity of the benchmarking results (Viotti & Kinderkhedia, 2022). The measurement of encode and decode was performed using the time.Now() function, while the payload size was calculated based on the length of the serialization result using the len() function.

The measurement results are then processed using statistical parameters such as average (avg), median (p50), and the 95th percentile (p95) to obtain a more representative picture of the system's performance (Kaler, 2017).

Due to the extremely short execution time of serialization and deserialization operations for smaller datasets, some p50 and p95 values are displayed as 0.000 µs. This occurs because the measured execution times fall below the display precision used in the benchmark output and therefore are rounded to zero. These values should be interpreted as indicating very low processing overhead rather than the absence of execution time.

Statistical Analysis and Interpretation

Statistical analysis was performed using average (avg), median (p50), and 95th percentile (p95) metrics to compare the performance of JSON and MessagePack in terms of payload size, encoding latency, decoding latency, and End-to-end latency. The p95 metric was included to capture latency variability and high-load behavior that may not be reflected by average values alone.

RESULTS AND DISCUSSION

The results and discussion in this study are divided into two main parts. The first part discusses the measurement of internal serialization and deserialization, which includes payload size, encoding latency, and decoding latency. The second part discusses the measurement of HTTP-based End-to-end latency, which includes the processes of serialization, data transmission, and response deserialization.

Serialization and Deserialization of Internal Data

Based on the measurement results in Table 3, there is a performance difference between JSON and MessagePack across all data size variations. The comparison was made regarding payload size, encoding latency, decoding latency, and performance percentiles to evaluate the system's stability under various load conditions. The measurement results were then visualized in Figures 1 to 4 to facilitate the performance analysis of both serialization formats.

It should be noted that several p50 and p95 values for smaller datasets appear as 0.000 µs. This result does not indicate that the serialization or deserialization process required no execution time. Instead, it reflects the limitation of the displayed measurement precision when processing very small payloads. Therefore, the average latency values and the results obtained from larger datasets provide a more meaningful basis for performance comparison.

Table 3. Internal Testing Results

Size	Codec	Payload	ENC Avg	ENC p50	ENC p95	DEC avg	DEC p50	DEC p95
10	JSON	739B	5.000µs	0.000µs	0.000µs	15.292µs	0.000µs	0.000µs
	MessagePack	695B	2.853µs	0.000µs	0.000µs	2.040µs	0.000µs	0.000µs
100	JSON	6.88KB	20.308µs	0.000µs	0.000µs	92.016µs	0.000µs	109.900µs
	MessagePack	6.57KB	9.893µs	0.000µs	0.000µs	22.228µs	0.000µs	47.200µs
1000	JSON	69.28KB	150.831µs	0.000µs	126.100µs	761.800µs	0.000µs	9.594ms
	MessagePack	66.34KB	111.240µs	0.000µs	508.500µs	166.369µs	0.000µs	917.600µs
5000	JSON	350.53KB	977.185µs	0.000µs	10.067ms	4.110ms	3.999ms	10.143ms
	MessagePack	335.87KB	520.576µs	0.000µs	1.152ms	763.160µs	0.000µs	2.104ms

Based on Figure 2, MessagePack shows lower encoding latency compared to JSON across all data sizes. The performance difference becomes more apparent with larger payload sizes, particularly with 5000 records . This condition is influenced by the data representation mechanism of MessagePack, which uses a binary format, allowing the serialization process to be carried out with lower processing overhead compared to text-based JSON(Cummings, 2024; Huseynov et al., 2024). Additionally, the more compact binary representation results in a smaller amount of data being processed, making the encoding process more efficient.

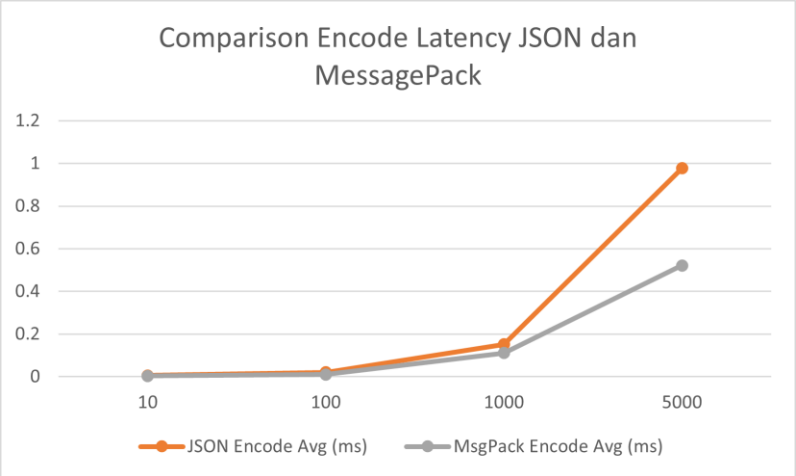

Figure 2. Chart comparison encode latency

Figure 2 demonstrates that encoding latency grows progressively as the number of records increases. For smaller datasets (10 and 100 records), the latency values remain relatively low, indicating that serialization overhead contributes only a minor portion of the total processing cost. As the dataset size reaches 1000 and 5000 records, the growth rate becomes substantially higher, suggesting that the computational workload of the serialization process increases with payload complexity. These results indicate that data volume plays a significant role in encoding performance and should be carefully considered when designing large-scale HTTP-based communication systems.

Based on Figure 3, MessagePack consistently shows faster decoding performance compared to JSON across all data sizes. The most significant difference is observed for larger datasets, where JSON experiences a considerable increase in decoding latency. This behavior occurs because JSON deserialization requires repeated character parsing, token interpretation, and type conversion before the original object structure can be reconstructed (Lu et al., 2024). In contrast, MessagePack uses a compact binary representation that enables more direct mapping between serialized data and application data types. As a result, fewer processing steps are required during deserialization, reducing computational overhead and latency variability. This allows MessagePack to maintain smoother and more consistent decoding performance as payload size increases.

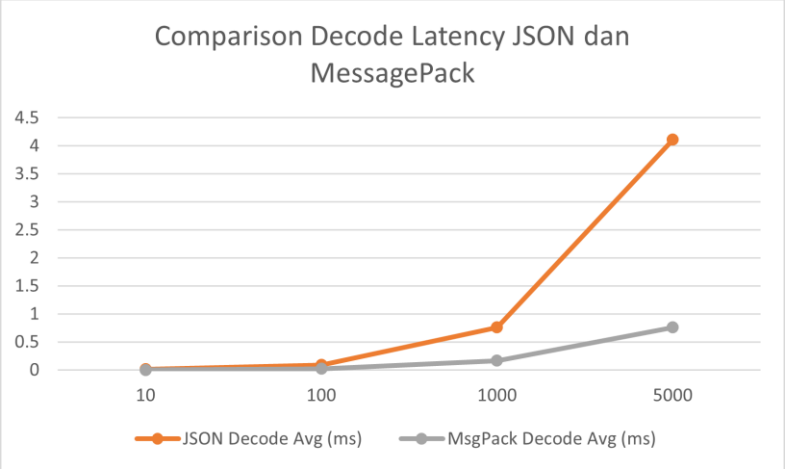

Figure 3. Chart comparison decode latency

Figure 3 demonstrates that decoding latency increases significantly as the dataset size grows, particularly for large payloads. While the latency values remain relatively low for datasets containing 10 and 100 records, a substantial increase can be observed when the dataset size reaches 1000 and 5000 records. This trend indicates that the deserialization process becomes increasingly demanding as the complexity and volume of the transmitted data increase. The sharp rise in latency at larger dataset sizes suggests that decoding operations are more sensitive to payload growth, making data size an important factor affecting the overall performance of HTTP-based communication systems.

Based on Figure 4, the p95 encode latency of JSON increases significantly for larger dataset sizes compared to MessagePack, indicating lower performance stability under high-load conditions. The 95th percentile (p95) metric represents latency behavior under near worst-case scenarios and provides insight into performance consistency beyond average values. Compared to decoding operations shown in Figure 5, encoding generally requires less processing time because it mainly converts existing in-memory objects into a serialized representation. In contrast, decoding involves additional operations such as data parsing, type identification, memory allocation, and object reconstruction. These extra computational steps make decoding more complex and contribute to higher latency values, particularly when processing larger payloads.

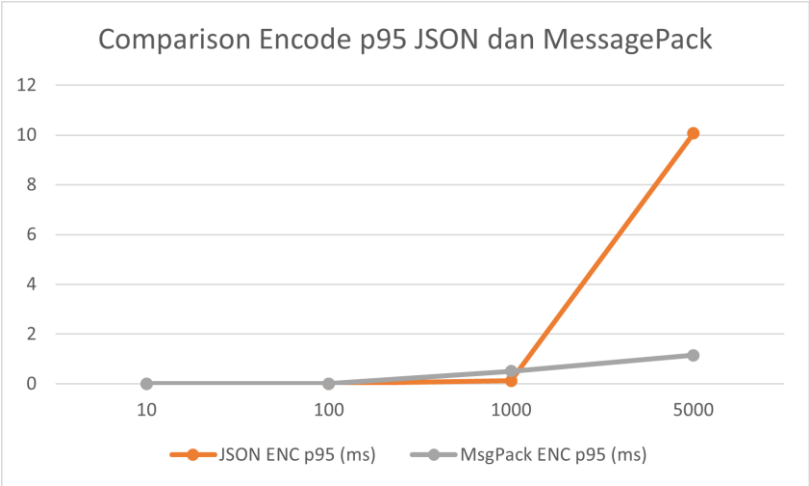

Figure 4. Encode Latency p95 Comparison

Figure 4 shows that the p95 encoding latency remains relatively stable for small and medium-sized datasets but increases sharply when the dataset reaches 5000 records. This behavior indicates that larger payloads introduce greater variability in encoding performance, leading to

occasional high-latency events. The difference between average latency and p95 latency becomes more pronounced at larger data sizes, suggesting that average values alone may not fully capture system behavior under peak processing conditions. Consequently, p95 measurements provide additional insight into the reliability and consistency of serialization performance in large-scale communication scenarios.

Based on Figure 5, the p95 decode latency of JSON increases substantially as the dataset size grows, while MessagePack exhibits a more gradual increase. The performance gap remains relatively small at 100 records because the latency distribution of both serialization formats is still concentrated within a narrow range, resulting in similar p95 values. However, when the dataset size reaches 1000 and 5000 records, the latency distribution of JSON becomes more dispersed, leading to a greater number of high-latency observations captured by the p95 metric. In contrast, MessagePack maintains a more compact latency distribution with lower variability. From a statistical perspective, these results indicate that larger payload sizes amplify latency fluctuations in JSON, causing the upper-tail latency represented by p95 to increase much faster than in MessagePack.

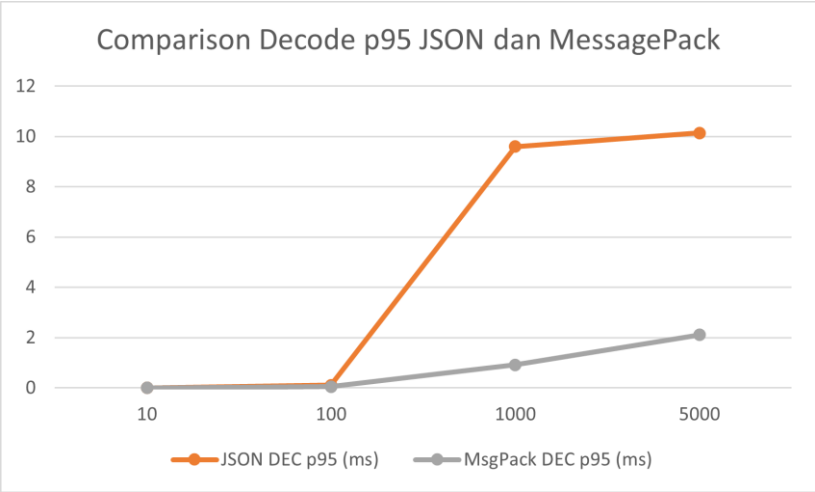

Figure 5. Chart decode p95

Figure 5 demonstrates that the growth of p95 decoding latency is substantially higher than the p95 encoding latency observed in Figure 4, particularly at the dataset size of 1000 records. This difference occurs because decoding involves additional computational processes beyond those required during encoding. During deserialization, the system must parse the incoming data, identify data types, allocate memory, and reconstruct application objects. These operations become increasingly expensive as payload size grows. In contrast, encoding mainly converts existing in-memory objects into a serialized representation, resulting in lower computational overhead. Consequently, the performance gap between JSON and MessagePack becomes more pronounced during decoding, especially for medium to large payload sizes.

End-to-end latency Measurement

After conducting internal serialization and deserialization testing, the next stage is measuring End-to-end latency based on HTTP. This test aims to determine the overall system performance, starting from the encoding process, sending the HTTP request, server processing, to the decoding response process back on the client side. The difference in End-to-end latency performance between JSON and MessagePack is influenced by the characteristics of their data representations. JSON uses a text-based format, so the serialization and deserialization processes require repeated character parsing. In contrast, MessagePack uses a more compact binary representation, allowing the encoding and decoding processes to be carried out with lower processing overhead.

Additionally, the smaller payload size of MessagePack makes the data transmission process over HTTP more efficient, especially for large data sizes (Zahra & Ardiansyah, 2026; Olufemi et al., 2023). The results of the End-to-end latency measurements between JSON and MessagePack are shown in Table 4 and will be visualized in figures 6 and 7.

Table 4. End-to-end latency Testing Result

SIZE	CODEC	REQ PAYLOAD	E2E avg	E2E p50	E2E p95
10	JSON	739 B	298.362 µs	510.800 µs	568.100 µs
	MsgPack	695 B	279.679 µs	31.400 µs	574.900 µs
100	JSON	6.88 KB	472.855 µs	537.300 µs	597.400 µs
	MsgPack	6.57 KB	293.068 µs	507.200 µs	582.000 µs
1000	JSON	69.28 KB	2.581 ms	2.325 ms	3.413 ms
	MsgPack	66.34 KB	1.152 ms	1.107 ms	1.852 ms
5000	JSON	350.53 KB	13.269 ms	13.141 ms	15.057 ms
	MsgPack	335.87 KB	3.894 ms	3.832 ms	4.675 ms

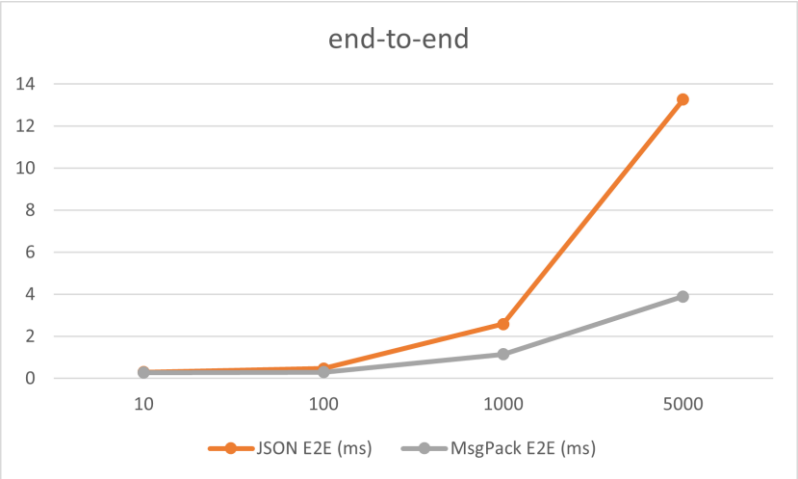

Figure 6. Chart End-to-end

Figure 6 illustrates the average End-to-end latency observed for JSON and MessagePack across different dataset sizes. For small datasets containing 10 and 100 records, the latency difference between the two serialization formats remains relatively small. However, as the dataset size increases to 1000 and 5000 records, the performance gap becomes substantially larger. This trend indicates that the impact of serialization format selection becomes increasingly significant under larger workloads. The results further demonstrate that MessagePack scales more efficiently than JSON as payload size grows, resulting in lower overall communication latency in HTTP-based systems.

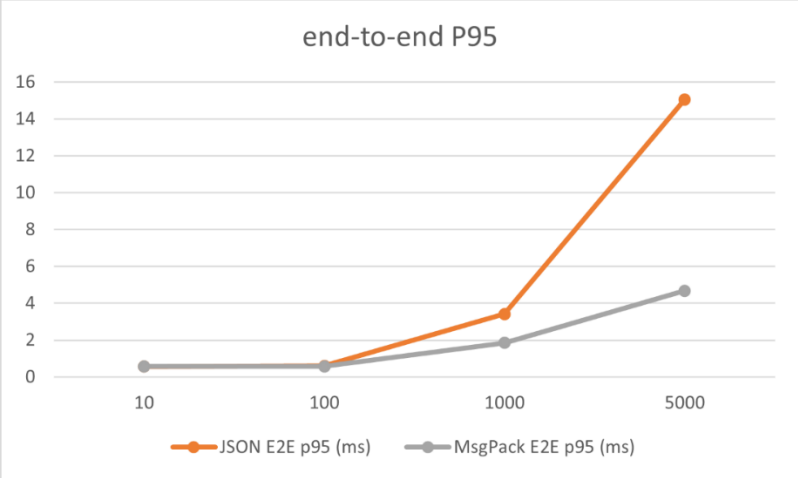

Figure 7. Chart End-to-end p95

Figure 7 presents the p95 End-to-end latency results, which reflect system performance under near worst-case operating conditions. The results show that both serialization formats exhibit similar p95 latency values for small datasets, indicating comparable performance consistency under light workloads. However, as the dataset size increases, the p95 latency of JSON grows much faster than that of MessagePack. This pattern suggests that larger payloads introduce greater latency variability for JSON, resulting in more frequent high-latency events. In contrast, MessagePack maintains lower p95 values, indicating more consistent performance and better reliability when processing large-scale data exchanges.

Comparative Analysis with Previous Research

To evaluate the contribution of the present work, the results are compared with several previous studies investigating the performance of serialization formats. As shown in Table 5, prior research consistently reported that binary serialization formats achieve smaller payload sizes and lower processing overhead than JSON. However, most studies focused on payload efficiency, serialization performance, or specific application domains. In contrast, the present study extends the analysis by evaluating encoding latency, decoding latency, payload size, and End-to-end latency HTTP communication performance under controlled benchmarking conditions using the Go programming language.

Table 5. Comparative Analysis

Study	Method	Dataset/Environment	Metrics	Main Findings
Viotti & Kinderkhedia (2022)	Benchmarking	Multiple serialization formats	Payload size, efficiency	Binary formats produced smaller payloads than JSON
Maltsev & Amin (2024)	Inter-service benchmark	Distributed service environment	Latency	Serialization format significantly affected latency
Cummings et al. (2024)	Experimental comparison of serialization formats	IoT management framework using JSON, Protocol Buffers, and FlatBuffers	Serialization time, deserialization time, payload size	Binary serialization formats reduced processing overhead and produced more compact payloads than JSON.
This Research	HTTP End-to-end benchmark	Go, 10–5000 records	Payload, Encode, Decode, E2E	MessagePack consistently outperformed JSON

The findings of this study are consistent with previous research indicating that binary serialization formats generally outperform JSON in terms of payload efficiency and processing performance. (Viotti & Kinderkhedia, 2022) reported that binary formats produce smaller payload sizes than JSON, while (Maltsev & Amin, 2024) demonstrated that serialization format selection significantly affects communication latency. Similarly, (Cummings, 2024) found that binary serialization formats reduce serialization overhead and improve processing efficiency. The results obtained in this study further support these observations, showing that MessagePack consistently achieved lower encoding latency, decoding latency, and End-to-end latency than JSON across all tested dataset sizes. Furthermore, by evaluating complete HTTP-based communication workflows using datasets ranging from 10 to 5000 records, this study provides additional evidence that the performance advantages of MessagePack become increasingly pronounced as payload size grows.

The results indicate that the choice of serialization format can significantly affect HTTP communication performance, particularly when large payloads are exchanged between services. The lower payload size and reduced processing overhead of MessagePack contribute to shorter response times and more efficient bandwidth utilization. These characteristics make MessagePack particularly suitable for high-traffic APIs, microservices architectures, and distributed systems where communication latency and resource efficiency are critical factors.

RESEARCH LIMITATIONS

This research still has limitations because the testing was conducted in a localhost environment, which does not fully represent real network conditions. Additionally, the testing did not involve a large number of concurrent requests or variations in network conditions such as packet loss and internet latency. Future research can be developed by comparing MessagePack with other serialization formats such as Protocol Buffers or Avro. Additionally, testing can be conducted on cloud-based microservices architectures with higher concurrent requests.

CONCLUSION

Based on the experimental results, MessagePack demonstrated better performance than JSON across all evaluated metrics, including payload size, encoding latency, decoding latency, and End-to-end latency. MessagePack consistently generated smaller payloads and lower processing overhead due to its compact binary representation. The performance differences became increasingly significant as the dataset size increased. For the largest dataset containing 5000 records, MessagePack reduced End-to-end latency from 13.269 ms to 3.894 ms and significantly improved decoding performance compared to JSON. These findings indicate that MessagePack is a suitable alternative for HTTP-based communication systems that require low latency, bandwidth efficiency, and high performance. Nevertheless, JSON remains advantageous in terms of readability, interoperability, and ease of debugging, making the choice of serialization format dependent on specific system requirements and application contexts. This study contributes to the empirical evaluation of serialization formats by providing an End-to-end latency HTTP benchmarking framework implemented in Go across multiple dataset scales. However, because the experiments were conducted under controlled localhost conditions, future research may extend this work by evaluating serialization performance under real network environments, varying levels of concurrency, and alternative communication protocols to provide a broader understanding of serialization behavior in practical distributed systems.

ACKNOWLEDGEMENTS

The authors would like to express their gratitude to Universitas Nahdlatul Ulama Blitar for supporting this research. The authors also thank the supervisors and all parties who contributed to the completion of this study.

REFERENCES

Araque, S. M., Lozada-Martínez, I. D., Papadopoulos, G. Z., Montavont, N., & Toutain, L. (2023). Yet Another Compact Time Series Data Representation Using CBOR Templates

(YACTS). In *Sensors*. https://doi.org/10.3390/s23115124

Araque, S. M., Martinez, I., Papadopoulos, G. Z., Montavont, N., & Toutain, L. (2022). Toward a Standard Time Series Representation for IoT based on CBOR Templates. *2022 Global Information Infrastructure and Networking Symposium (GIIS)*, 13–19. https://doi.org/10.1109/GIIS56506.2022.9936910

Ardiansyah, H., & Fatwanto, A. (2022). Comparison of Memory Usage Between REST API in Javascript and Golang. In *Matrik Jurnal Manajemen Teknik Informatika Dan Rekayasa Komputer*. https://doi.org/10.30812/matrik.v22i1.1325

Balalayeva, E., Marchenko, I., & Serhiienko, A. (2024). Investigation of the Performance Efficiency of C# Programming Language Data Serializers using the Developed Software Product for Testing. *2024 IEEE 19th International Conference on Computer Science and Information Technologies (CSIT)*, 1–4.

Bermbach, D., & Wittern, E. (2020). *Benchmarking Web API Quality – Revisited*. *19*, 603–646. https://doi.org/10.13052/jwe1540-9589.19563

Cummings, N. (2024). Streaming Technologies and Serialization Protocols : Empirical Performance Analysis. *IEEE Access*, *12*(October), 158025–158039. https://doi.org/10.1109/ACCESS.2024.3486054

Huseynov, K., Cakir, L. V., Al-Shareeda, S., Özdem, M., & Canberk, B. (2024). A data serialization-based framework for efficient IoT management. *2024 IEEE 10th World Forum on Internet of Things (WF-IoT)*, 707–712.

Kaler, T. (2017). *Optimal Reissue Policies for Reducing Tail Latency*. 195–206.

Keiser, J. (2024). *On-demand JSON : A better way to parse documents ? November 2023*, 1074–1086. https://doi.org/10.1002/spe.3313

Langdale, G., & Lemire, D. (2019). Parsing gigabytes of JSON per second. *The VLDB Journal*, *28*(6), 941–960.

Lu, F., Wei, X., Huang, Z., Chen, R., Wu, M., & Chen, H. (2024). *Serialization / Deserialization-free State Transfer in Serverless Workflows*. 132–147. https://doi.org/10.1145/3627703.3629568

Maltsev, E., & Amin, R. U. (2024). Impact of Serialization Format on Inter-Service Latency. *Advances in Cyber-Physical Systems*, *9*(2), 89–94. https://doi.org/10.23939/acps2024.02.089

Marcelino, C., Pusztai, T., & Nastic, S. (2025). Roadrunner: Accelerating Data Delivery to WebAssembly-Based Serverless Functions. *Proceedings of the 26th International Middleware Conference*, 354–368.

Myastovskiy, T. (2024). *Evaluating the Performance of Serialization Protocols in Apache Kafka*.

Olufemi, O. I., Ukommi, U., & Ubom, E. (2023). Comparative analysis of transceiver payload size impact on the performance of LoRa-based sensor node. *Sleep*, *31050*, 45–500.

Pargaonkar, S. (2023). *A Comprehensive Review of Performance Testing Methodologies and Best Practices : Software Quality Engineering*. *12*(8), 2008–2014. https://doi.org/10.21275/SR23822111402

Proos, D. P. (2020). *Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV*. *June*.

Sutisnawati, Y., Maryati, M., & Setiyadi, A. (2025). *Pengembangan API Berbasis JSON pada Sistem Koperasi Berbasis Web*. *5*(2), 303–316.

Valiveti, S. S. (2025). Low Latency Web APIs in High Transaction Systems Design and Benchmarking. *International Journal of Computational and Experimental Science and Engineering*, *11*(3).

Viotti, J. C., & Kinderkhedia, M. (2022). *A Benchmark of JSON-compatible Binary Serialization Specifications*. https://doi.org/10.48550/arxiv.2201.03051

Yuan, C., Du, J., Yue, M., & Ma, T. (2020). The Design of Large Scale IP Address and Port Scanning Tool. In *Sensors*. https://doi.org/10.3390/s20164423

Yudanto, B. A., Laudri, S. P., Felina, I., Rizqi, M., Abinovan, A., Fadhil, M., Fathir, R., Putra, R., Ulfa, D. F., Salzi, W. F., Nuzulah, R., Studi, P., Informatika, T., Gedong, K., Rebo, P., & Timur, J. (2025). *IMPLEMENTASI REST API MENGGUNAKAN BAHASA GO*. *06*(02), 384–393.

Zahra, V. A., & Ardiansyah, R. (2026). HTTP/3 vs HTTP/2: A Comparative Analysis of Latency and Throughput in RESTful API Transactions Under Varying Concurrency, Payload, and Degraded Network Conditions. *Journal of Applied Informatics and Computing*, *10*(2), 1291–1303.