Journal of Engineering and Technology for Industrial Applications

ISSN ONLINE: 2447-0228

ITEGAM-JETIA

Manaus, v.12 n.57, p. 718-727. January/February, 2026.
DOI: https://doi.org/10.5935/jetia. v12i57.3040

RESEARCH ARTICLE OPEN ACCESS

PERFORMANCE EVALUATION OF JSON LIBRARIES IN KOTLIN SERVER ENVIRONMENTS: JACKSON, GSON, AND KOTLINX SERIALIZATION

Achmad Arwan*[1], Julia Nur Fajrina[2] and Eriq Muhammad Adam Jonemaro[3]

[1, 2, 3]Department of Informatic Engineering, Faculty of Computer Science, Brawijaya University, Malang, Indonesia.

[1]https://orcid.org/0000-0002-4146-363X, [2]https://orcid.org/0009-0008-4817-3599, [3]https://orcid.org/0000-0001-8315-3210

Email: *arwan@ub.ac.id, juliarina@student.ub.ac.id, eriq.adams@ub.ac.id

ARTICLE INFO

Article History
Received: December 2, 2025
Revised: January 10, 2026
Accepted: January 15, 2026
Published: February 28, 2026

Keywords:
Kotlin,
JSON libraries,
Jackson,
Server-side performance,
JSON processing.

ABSTRACT

The widespread adoption of Kotlin for server-side development has emphasized the importance of efficient JSON processing. This study aimed to empirically evaluate and compare the performance of three JSON libraries—Jackson, Gson, and KotlinX Serialization—in Kotlin-based server environments. Using structured experiments, the research measured execution time, memory utilization, and CPU consumption during parsing, serialization, and deserialization tasks involving nested JSON data with incremental complexity. Statistical analyses confirmed significant performance differences among libraries. KotlinX Serialization consistently outperformed Jackson and Gson across execution speed, memory efficiency, and CPU utilization, primarily due to its compile-time code generation approach. Jackson showed superior memory management and efficient serialization for smaller datasets, while Gson exhibited lower overall performance. These results offer critical guidance for developers selecting appropriate JSON libraries, highlighting KotlinX Serialization's suitability for high-performance applications. The findings significantly enhance the current understanding of JSON processing efficiency and suggest avenues for further investigation into additional libraries and real-world application contexts.

I. INTRODUCTION

The rapid evolution of software technology has significantly elevated the importance of server-side development, as backend systems have become crucial in ensuring scalability, performance, and robustness of modern applications. Kotlin, a statically-typed programming language introduced by JetBrains, has steadily gained popularity as an efficient alternative to Java, particularly for server-side programming. Kotlin's concise syntax, enhanced type safety, interoperability with Java, and modern programming paradigms such as coroutines have substantially increased its adoption among developers [1]. This growth has been notably supported by frameworks explicitly tailored for Kotlin, such as Ktor, and by mainstream Java frameworks like Spring Boot, which have adopted official support for Kotlin [2]. According to the JetBrains Developer Ecosystem Report (2023), approximately 52% of Kotlin developers use Kotlin for backend development, signifying an upward trend in its acceptance within the industry [2].

Leading global enterprises including Amazon, Atlassian, Adobe, and Mercedes-Benz.io have embraced Kotlin within their backend infrastructure, particularly for microservices and enterprise-level applications, further validating its suitability for high-performance environments [3]. A crucial aspect of server-side development involves effective data communication, where JSON (JavaScript Object Notation) has become the standard format due to its lightweight nature, ease of parsing, human readability, and compatibility across diverse platforms[4],[5]. JSON's semi-structured nature makes it adaptable to various application scenarios but also requires efficient processing mechanisms, including parsing, serialization, and deserialization, to transform JSON data into usable native objects within server applications. These processing tasks significantly impact the overall performance of a server, particularly in contexts involving high volumes of concurrent data exchanges [6]. Efficient JSON processing is thus imperative, highlighting the importance of selecting suitable JSON libraries capable of optimizing performance and resource utilization in Kotlin-based server environments.

Despite the availability of numerous JSON processing libraries, empirical comparisons that specifically focus on Kotlin server environments remain relatively scarce. The core research problem arises from the necessity to evaluate performance discrepancies among available JSON libraries under realistic workloads typical of Kotlin server applications. Different libraries approach JSON processing through varying computational strategies. Prominent among these are Jackson, Gson, and KotlinX Serialization, each employing distinct methodologies that directly influence their runtime efficiency. Jackson and Gson predominantly rely on reflection, dynamically inspecting objects at runtime to facilitate JSON transformations[7],[8]. Conversely, KotlinX Serialization employs a fundamentally different strategy, generating serialization logic at compile-time through compiler plugins, thus reducing runtime overhead [9].

Addressing this central problem requires an approach encompassing empirical evaluation, systematic experimentation, and rigorous statistical analysis. General solutions often involve benchmarking libraries against multiple performance metrics, including execution time, memory consumption, and CPU usage, under various data complexity scenarios. However, such evaluations must also explicitly consider statistical significance to ensure robust, scientifically valid conclusions. Existing literature typically lacks this methodological rigor or generalizes findings from non-Kotlin environments, reducing their relevance to the Kotlin ecosystem. Previous research provides insights into JSON processing performance, although these studies differ substantially in terms of contexts, methodologies, and goals.

Maeda examined the efficiency of various serialization formats, including XML, JSON, and binary formats, within Java-based applications, focusing on comparative metrics such as serialization speed and output size[10]. Vanura and Kriz further evaluated the performance of JSON, MessagePack, and Protocol Buffers across languages like Java, PHP, and JavaScript, emphasizing both speed and data compactness[11]. Similarly, Dhalla conducted performance comparisons among native JSON parsers across multiple languages—Java, Python, .NET Core, JavaScript, and PHP—with emphasis on parsing efficiency, memory usage, and CPU consumption [5]. While valuable, these studies either omit Kotlin-specific implementations or neglect rigorous statistical testing, presenting limitations that hinder direct applicability to Kotlin-based server contexts.

Moreover, in studies specifically addressing Kotlin, KotlinX Serialization emerges as a promising solution explicitly designed to address Kotlin's inherent language features and runtime characteristics. Unlike Jackson and Gson, KotlinX Serialization utilizes compiler-generated code, providing native support for Kotlin's language-specific attributes such as data classes, default parameters, and null-safety guarantees, significantly reducing runtime reflection overhead and improving serialization performance[12]. This inherent compatibility and performance optimization through compile-time code generation positions KotlinX Serialization as an advantageous choice, particularly in scenarios demanding rapid processing and minimal resource usage. However, while KotlinX Serialization theoretically offers these advantages, empirical evidence supporting these claims in realistic server-side conditions remains insufficiently explored.

Previous research has generally tested JSON processing performance within generic or language-neutral environments, often neglecting Kotlin's specific runtime interactions and JVM integration nuances. Furthermore, the absence of comprehensive experimental scenarios involving complex, nested JSON structures, which closely resemble practical server-side data exchanges, limits the existing evidence's real-world applicability. Consequently, a clearly defined research gap emerges, characterized by inadequate empirical benchmarking of JSON libraries within the Kotlin server ecosystem, specifically regarding rigorous statistical testing and practical experimental scenarios involving nested and complex data structures. The existing literature demonstrates significant variations in performance among libraries across different environments and programming contexts, yet lacks specificity and methodological rigor for Kotlin-based implementations.

To address this gap, this study aims to empirically evaluate and compare the performance of three widely-used JSON libraries—Jackson, Gson, and KotlinX Serialization—in realistic server-side scenarios explicitly developed for Kotlin. The primary objective is to systematically assess and quantify each library's efficiency in parsing, serialization, and deserialization tasks involving nested JSON data of varying complexities. The experimental design incorporates multiple performance metrics, namely execution time, memory usage, and CPU consumption, measured under strictly controlled, reproducible conditions. Furthermore, the research methodology integrates comprehensive statistical analysis to validate the significance of observed performance differences, enhancing the robustness and scientific validity of the findings.

The novelty of this study resides in its dedicated Kotlin-specific focus, addressing the need for evidence-based insights applicable directly to Kotlin server-side development. By employing rigorous statistical methodologies and realistic, structured experimental scenarios, the research seeks to provide definitive, actionable insights to support informed decision-making among developers selecting JSON processing libraries. The scope of the study is explicitly confined to server-side JSON processing within Kotlin-based environments, excluding mobile and front-end applications, thereby offering targeted relevance and practical utility for Kotlin backend developers and researchers.

II. THEORETICAL REFERENCE

II.1 SERVER-SIDE DEVELOPMENT AND THE ROLE OF KOTLIN

Server-side development has become fundamental to modern software architecture, providing the backbone for scalability, performance optimization, and system robustness. Kotlin, a statically-typed programming language developed by JetBrains, has emerged as a compelling alternative to Java for backend systems. According to the JetBrains Developer Ecosystem Report (2023), approximately 66% of Kotlin developers utilize the language for Android and server-side applications, demonstrating substantial adoption in enterprise environments [13], [14].

Kotlin's design philosophy emphasizes concise syntax, enhanced type safety, and seamless interoperability with existing Java ecosystems. The language incorporates modern programming paradigms, particularly coroutines for asynchronous programming, which enable developers to write non-blocking code in a sequential style without the complexity of traditional callback mechanisms. Spring Boot, one of the most widely adopted backend frameworks, has provided official Kotlin support since 2017, offering first-class coroutine support, Kotlin-specific DSLs, and enhanced null-safety features.

Similarly, Ktor, a lightweight asynchronous framework built by JetBrains specifically for Kotlin, leverages coroutines for high scalability and provides intuitive Domain-Specific Language (DSL) capabilities for server-side development. Major global enterprises including Amazon, Atlassian, Adobe, and Mercedes-Benz have integrated Kotlin into their backend infrastructure, particularly for microservices and enterprise-level applications, validating its suitability for production environments requiring high performance and reliability. Type safety represents a fundamental property of programming languages that ensures variables access only authorized memory locations in well-defined, permissible ways, thereby preventing invalid operations on objects.

Statically-typed languages like Kotlin perform type checking at compile-time, enabling compilers to optimize code execution and memory usage while catching type errors before runtime. Kotlin's type system incorporates null-safety as a first-class language feature, distinguishing between nullable and non-nullable types at the type-system level. This design choice eliminates entire categories of runtime errors common in Java, particularly NullPointerException occurrences. For JSON serialization libraries, native support for Kotlin's type system features—as provided by KotlinX Serialization—theoretically reduces impedance mismatch between language semantics and serialization logic[15].

II.2 JSON AS THE STANDARD FOR DATA EXCHANGE

JavaScript Object Notation (JSON) has established itself as the de facto standard for data interchange in server-side development due to its lightweight structure, human readability, and cross-platform compatibility. JSON's semi-structured nature provides flexibility for representing complex data hierarchies while maintaining simplicity in parsing and generation. Data serialization, defined as the process of converting data objects into byte streams or character sequences for storage or transmission, forms a critical component of server-side operations. The reverse process, deserialization, reconstructs the original data structure from the serialized format.

In distributed systems and high-performance computing environments, efficient serialization directly impacts system throughput, latency, and resource utilization[16], JSON serialization involves encoding complex data structures while preserving information such as type, order, and relationships between nested elements. The performance characteristics of JSON processing—including parsing speed, memory consumption, and CPU overhead—significantly affect overall server performance, particularly in scenarios involving high-volume concurrent data exchanges typical of modern backend applications.

II.3 PERFORMANCE METRICS IN SOFTWARE SYSTEMS

Performance evaluation of software systems relies on standardized metrics that quantify computational efficiency and resource utilization. The three primary performance dimensions assessed in this study are Execution Time, Memory Usage and CPU Consumption. Execution Time (CPU Time) represents the total duration a CPU spends computing a specific task, excluding time spent on I/O operations or executing other processes. CPU execution time can be calculated as the product of CPU clock cycles and clock cycle time, or equivalently, as the ratio of instruction count and clock rate. This metric directly measures computational efficiency and is inversely proportional to system performance.

Memory Usage quantifies the amount of RAM consumed during program execution, typically measured in bytes, kilobytes, or megabytes. Memory utilization patterns significantly impact system scalability, particularly in server environments handling concurrent requests. Excessive memory consumption can trigger garbage collection overhead in JVM-based systems, indirectly affecting execution time. CPU Consumption is the Measures the percentage of CPU resources utilized during execution, providing insight into processor workload and potential bottlenecks. High CPU utilization may indicate computationally intensive operations, while low utilization might suggest I/O-bound or inefficiently parallelized code.

II.4 STATISTICAL SIGNIFICANCE TESTING IN PERFORMANCE EVALUATION

Statistical significance testing provides the methodological framework for determining whether observed performance differences between systems reflect genuine effects rather than random variation. The Neyman-Pearson hypothesis testing paradigm, widely adopted in empirical software engineering, involves formulating a null hypothesis (typically asserting no difference between conditions) and an alternative hypothesis, then calculating the probability (p-value) of obtaining the observed data assuming the null hypothesis is true [17]. A critical consideration in performance benchmarking is the relationship between sample size and statistical power. As sample size increases, the probability of detecting true effects (statistical power) increases while the risk of Type II errors (false negatives) decreases.

However, with very large samples, even trivial effect sizes may achieve statistical significance, necessitating consideration of practical significance in addition to statistical significance [17].The choice of significance threshold (α level, commonly set at 0.05) represents a balance between Type I errors (false positives) and Type II errors (false negatives). Some researchers argue that exploratory studies with small sample sizes may warrant more liberal thresholds (e.g., $\alpha = 0.20$), while confirmatory studies with large samples should employ more conservative thresholds. Beyond p-values, effect size measures quantify the magnitude of observed differences, providing context for practical importance independent of sample size[17].

II.5 PRIOR RESEARCH ON SERIALIZATION PERFORMANCE

Previous empirical studies have examined serialization format performance across various programming language contexts, though comprehensive Kotlin-specific evaluations remain limited. Maeda (2012) compared object serialization libraries across XML, JSON, and binary formats in Java-based applications, focusing on serialization speed and output size as primary metrics. Vanura and Kriz evaluated JSON, MessagePack, and Protocol Buffers across Java, PHP, and JavaScript, emphasizing speed and data compactness but omitting Kotlin implementations and rigorous statistical validation [18]. Conducted performance analysis of native JSON parsers across five programming languages—Java, Python, PHP, .NET Core, and JavaScript—measuring parsing speed, memory usage, and CPU consumption for JSON documents with varying nesting depth and key-value pair counts. Their findings demonstrated substantial performance variation across language environments, though Kotlin was not included in the comparative analysis [19].

Studies specifically examining MessagePack, Protocol Buffers, and other binary serialization formats have consistently found trade-offs between serialization speed, deserialization speed, and payload size. Protocol Buffers typically achieves faster serialization and smaller payloads compared to JSON, while MessagePack offers balanced performance across multiple dimensions. However, these studies generally employ Java-centric implementations without considering Kotlin-specific libraries or language features. Research addressing Kotlin serialization specifically highlights KotlinX Serialization's compile-time code generation approach as theoretically advantageous for performance. However, empirical validation under realistic server-side workloads with rigorous statistical analysis and comprehensive performance metrics remains underrepresented in the literature. This gap motivates the present study's focus on systematic, statistically validated comparison of Jackson, Gson, and KotlinX Serialization within Kotlin server environments [20].

III. MATERIALS AND METHODS

Write This study adopts a quantitative, empirical research approach to systematically evaluate and compare the performance of JSON libraries—Jackson, Gson, and KotlinX Serialization—within a Kotlin-based server environment. A between-subject experimental design was utilized, wherein each JSON library was tested independently against identical JSON data sets. This experimental approach was chosen to ensure unbiased performance metrics specific to each library, allowing precise analysis of differences in execution time, memory utilization, and CPU consumption under controlled conditions. The research diagram shown on Figure 1.

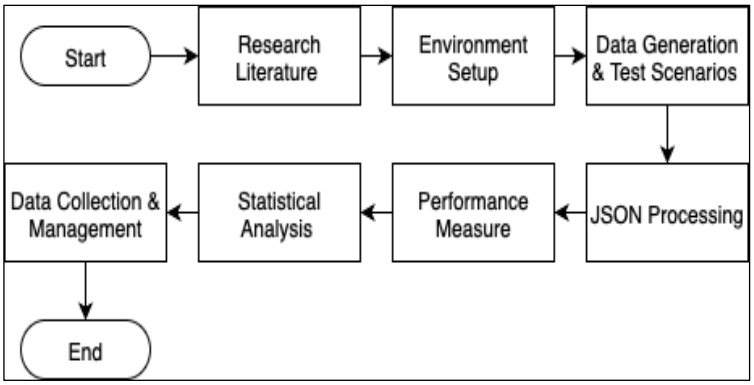

Figure 1: Research Methodology.
Source: Authors, (2026).

III.1 EXPERIMENTAL ENVIRONMENT

The experimental setup consisted of a server developed using the Kotlin programming language, facilitated by the Ktor framework, to ensure relevance to real-world server implementations. Docker containers were employed to guarantee consistency, reproducibility, and isolation of experimental environments across multiple test runs. The hardware specifications used in this research included an HP Pavilion 14 laptop with an AMD Ryzen 5 5500U processor, 16 GB RAM, and Windows 11 64-bit operating system. Precise measurement of execution time was obtained using the System.nanoTime() method. Memory utilization was assessed by comparing heap memory usage before and after each process through the Java MemoryMXBean API, and CPU consumption was monitored using OperatingSystemMXBean.

III.2 DATA GENERATION AND TEST SCENARIOS

Test data were generated using an adapted version of an algorithm developed by [5] to create structured, nested JSON files suitable for benchmarking JSON libraries. The data comprised ten distinct JSON files, each characterized by a fixed nesting depth of 10 levels, with increasing complexity achieved by incrementing the number of key-value pairs at each nesting level from 10 to 100, in steps of 10. Each file thus represented progressively larger and more complex data structures, reflective of realistic server-side processing scenarios. At the deepest nesting level, a structured JSON array containing additional nested objects extended the complexity further to an effective depth of 20 levels, emulating real-world data complexity. The structure data shown on Figure 2. The file size and the number of key shown on Table 1.

III.3 JSON PROCESSING PROCEDURES

The evaluation encompassed three primary JSON data processing tasks: parsing, serialization, and deserialization. Parsing refers explicitly to the transformation of raw JSON strings into intermediary data structures (JsonElement), without type-specific conversions. In contrast, deserialization converts JSON strings directly into typed Kotlin objects, while serialization transforms Kotlin objects back into JSON strings. These definitions align with distinctions drawn by [21] in the KotlinX Serialization documentation, ensuring clarity and consistency across all processing tasks.

Figure 2: JSON Data Structure.
Source: Authors, (2026).

Table 1: Key and file Size.

No	*Key Number*	*File Size* (KB)
1	10	58
2	20	117
3	30	176
4	40	236
5	50	295
6	60	355
7	70	414
8	80	474
9	90	533
10	100	592

Source: Authors, (2026).

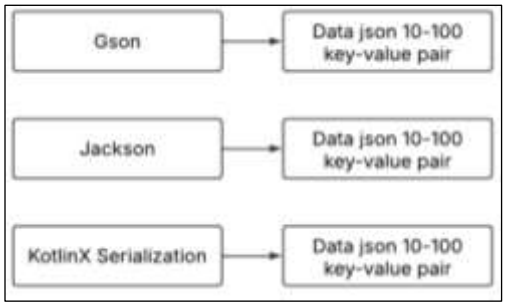

Figure 3: Library and Number Experiments.
Source: Authors, (2026).

Each processing task was performed in two stages to enhance accuracy and reliability: an initial warm-up phase followed by a main measurement phase. The warm-up phase consisted of 500 initial iterations intended to stabilize the JVM environment, mitigating the effects of Just-In-Time (JIT) compilation and runtime optimizations. Subsequently, the main measurement phase involved 5,000 experimental runs, each comprising 50 repetitions of parsing, serialization, or deserialization per test file. Performance metrics were averaged across runs to yield stable and representative results.

III.4 PERFORMANCE METRICS

Three key performance indicators were systematically measured: execution time, memory usage, and CPU consumption. Execution time, recorded in milliseconds (ms), was captured precisely to evaluate processing speed. Memory usage, assessed in megabytes (MB), provided insights into the libraries' efficiency in resource utilization. CPU consumption, expressed as a percentage, indicated computational load under identical processing tasks, offering a comprehensive view of resource efficiency. All metrics were recorded consistently across parsing, serialization, and deserialization processes to enable direct comparative analysis.

III.5 STATISTICAL ANALYSIS

Rigorous statistical analyses were conducted to ensure the reliability and validity of the performance comparisons. Initially, data were tested for normality using the Kolmogorov-Smirnov test to establish appropriate subsequent analytical methods. Due to the non-normal distribution of performance metrics, non-parametric statistical methods were selected for further analysis. Specifically, the Kruskal-Wallis test, a non-parametric alternative to ANOVA, was employed to detect statistically significant differences among library performances across all experimental conditions. A significance level (α) of 0.05 was chosen to determine the statistical significance of observed differences. Following the identification of significant differences via the Kruskal-Wallis test, Dunn's post hoc test was used for pairwise comparisons to pinpoint precisely where significant differences existed between individual library pairs[22].

III.6 DATA COLLECTION AND MANAGEMENT

Performance data obtained from experimental runs were systematically logged and stored in CSV format to facilitate structured data management and subsequent statistical analysis. Each CSV entry contained clearly labelled and timestamped metrics for execution time, memory utilization, and CPU consumption, ensuring traceability and reproducibility of data collection processes. Data integrity and consistency were verified through regular validation checks performed throughout the experimental period.

III.7 LIMITATIONS AND SCOPE

The methodological approach of this study was deliberately limited to evaluating three widely-adopted JSON libraries within a Kotlin-based server context. This decision was guided by the prevalence and relevance of these libraries within Kotlin's developer ecosystem. The scope excluded libraries not explicitly designed or widely used for Kotlin environments and did not extend to front-end or mobile applications. Furthermore, testing was confined to controlled laboratory conditions, and results may vary in different hardware or operating environments.

Future research could explore broader comparisons involving additional libraries, other programming environments, or real-world deployment scenarios. Overall, the structured and systematic approach adopted in this study, encompassing rigorous experimental procedures and comprehensive statistical analysis, was designed to yield valid, reliable, and practically relevant insights into the comparative performance of Jackson, Gson, and KotlinX Serialization libraries within Kotlin-based server environments. This methodology addresses a clear research gap, providing robust empirical evidence to guide informed decision-making among software developers and system architects.

IV. RESULTS AND DISCUSSIONS

IV.1 OVERVIEW OF EXPERIMENTAL RESULTS

This section presents the results obtained from rigorous empirical testing of three JSON libraries—Jackson, Gson, and KotlinX Serialization—within a Kotlin-based server environment. The performance metrics examined include execution time, memory usage, and CPU consumption, measured consistently across parsing, serialization, and deserialization processes. Each process was evaluated using structured nested JSON datasets, progressively increasing in complexity by incrementing key-value pairs from 10 to 100 per nesting level. Detailed statistical analyses were conducted to ensure the reliability and robustness of the findings.

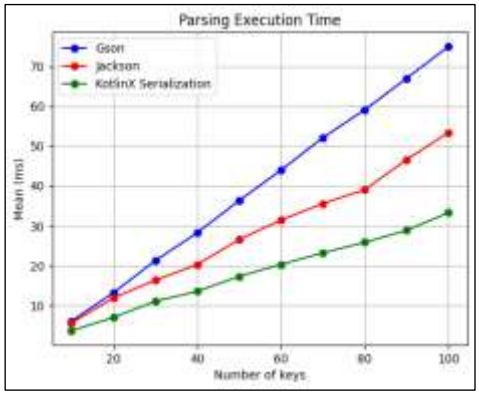

Figure 4: Parsing Performance.
Source: Authors, (2026).

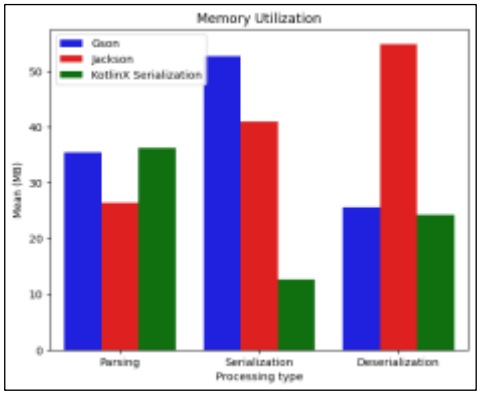

Figure 5: Memory Utilization.
Source: Authors, (2026).

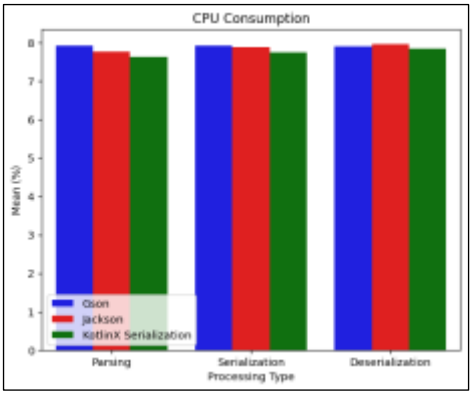

Figure 6: CPU Consumption.
Source: Authors, (2026).

IV.1.1 Parsing Performance

The parsing process involved transforming raw JSON strings into intermediate JSON data structures (JsonElement). The results of performance evaluation shown on Figure 4. Performance evaluation revealed distinct differences among the libraries tested. Specifically, KotlinX Serialization consistently exhibited the fastest execution times across all file complexities. For instance, with the smallest file (10 key-value pairs), KotlinX Serialization achieved an average execution time of 3.63 milliseconds, significantly outperforming Jackson (5.57 milliseconds) and Gson (6.03 milliseconds). As file complexity increased, performance advantages for KotlinX Serialization became increasingly pronounced, with a clear trend observed across all complexity levels. Memory utilization during parsing, serialize and deserialize depicted on Figure 5. Memory utilization during parsing showed a different pattern. Jackson demonstrated superior efficiency, consistently consuming less memory compared to both Gson and KotlinX Serialization.

For example, Jackson's average memory usage was approximately 26.35 megabytes, markedly lower than Gson's 35.46 megabytes and KotlinX Serialization's 36.27 megabytes across all tests. This efficiency advantage for Jackson remained consistent irrespective of file size or complexity. CPU consumption during parsing, serialize and deserialize depicted on Figure 6. CPU consumption data during parsing further emphasized KotlinX Serialization's performance advantage, albeit with narrower margins. KotlinX Serialization recorded the lowest CPU usage at an average of 7.01%, marginally better than Jackson (7.49%) and Gson (7.93%). This lower CPU consumption by KotlinX Serialization suggests better optimization in computational efficiency during parsing processes.

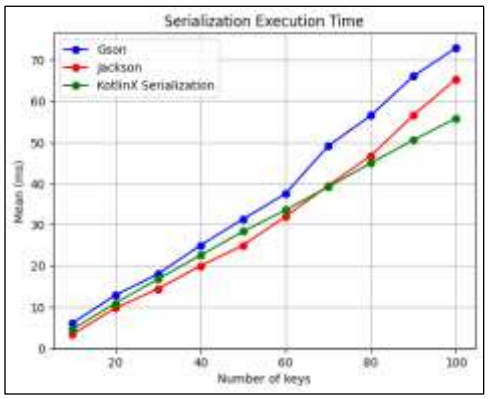

Figure 7: Serialization Performance.
Source: Authors, (2026).

IV.1.2 Serialization Performance

Serialization performance, involving the transformation of native Kotlin objects into JSON strings, presented contrasting results compared to parsing. The results of Serialization performance shown on Figure 7. Jackson emerged as the most efficient library regarding execution time for smaller and medium-sized files (10 to 60 key-value pairs). For instance, at the smallest file size (10 keys), Jackson completed serialization in an average of 3.36 milliseconds, significantly quicker than KotlinX Serialization's 4.56 milliseconds and Gson's 6.08 milliseconds. However, as file complexity grew (70 to 100 key-value pairs), KotlinX Serialization progressively outperformed Jackson and Gson, clearly demonstrating superior scalability for larger data sets.

Memory usage during serialization illustrated pronounced advantages for KotlinX Serialization across all file sizes (Figure 5). KotlinX Serialization consistently used significantly less memory, averaging 12.59 megabytes, compared to Jackson's 40.87 megabytes and Gson's 52.64 megabytes. This substantial efficiency in memory utilization highlights KotlinX Serialization's suitability for resource-constrained server environments. CPU consumption data indicated minimal variations between libraries, with KotlinX Serialization marginally outperforming others (depicted on Figure 6). It exhibited average CPU usage of 7.38%, slightly lower than Jackson (7.43%) and Gson (7.69%), signifying a minor yet consistent advantage in computational efficiency during serialization processes.

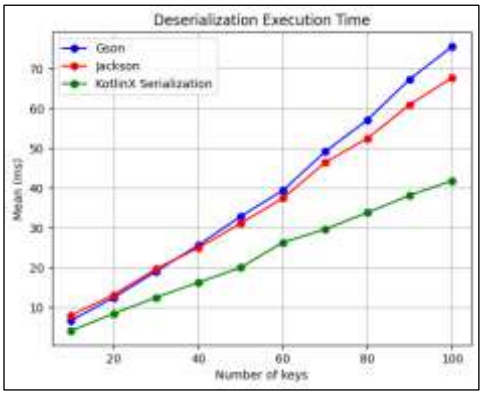

Figure 8: Deserialization Performance.
Source: Authors, (2026).

IV.1.3 Deserialization Performance

The deserialization process, converting JSON strings directly into typed Kotlin objects, underscored KotlinX Serialization's comprehensive performance superiority. The results of deserialization performance shown on Figure 8. Execution time results consistently favoured KotlinX Serialization across all file complexities, from simple to highly complex JSON structures. For instance, at the smallest complexity level (10 keys), KotlinX Serialization deserialized data in an average of 4.11 milliseconds, significantly faster than Gson (6.68 milliseconds) and Jackson (7.99 milliseconds). This advantage was maintained and increased proportionally with growing file complexities. Memory utilization during deserialization (Figure 5) also demonstrated KotlinX Serialization's efficiency, using significantly less memory on average (24.26 megabytes) compared to Gson (25.52 megabytes) and Jackson (54.78 megabytes). These findings reinforce KotlinX Serialization's optimal memory efficiency, making it particularly suited for large-scale server environments dealing with extensive JSON data.

Regarding CPU consumption during deserialization (depicted on Figure 6), KotlinX Serialization again demonstrated superior efficiency. It maintained the lowest CPU usage at an average of 7.44%, slightly outperforming Gson (7.69%) and Jackson (7.84%). These results underscore KotlinX Serialization's balanced approach to resource utilization, delivering consistent performance advantages across all evaluated metrics.

IV.1.4 Statistical Analysis

Wherever To substantiate the empirical findings, statistical analyses were rigorously conducted. Initial normality testing using the Kolmogorov-Smirnov method indicated that the data sets did not follow a normal distribution. Consequently, non-parametric statistical tests were employed to ensure valid and accurate comparative analyses. The Kruskal-Wallis test, a robust non-parametric method, identified statistically significant differences among the three libraries across all performance metrics and JSON processing tasks. All calculated p-values were significantly below the predetermined alpha level of 0.05, thus confirming meaningful performance differences among Jackson, Gson, and KotlinX Serialization. Post-hoc testing using Dunn's test provided detailed pairwise comparisons, clearly delineating significant performance differences between each library pair. These analyses consistently demonstrated that KotlinX Serialization significantly outperformed Jackson and Gson in execution time and CPU efficiency across parsing and deserialization tasks, while Jackson was superior in memory efficiency during parsing tasks.

IV.1.5 Summary Of Findings

The Overall, the comprehensive evaluation of JSON processing tasks—parsing, serialization, and deserialization—revealed clear and statistically significant performance differences among Jackson, Gson, and KotlinX Serialization. KotlinX Serialization consistently demonstrated superior execution speed and CPU efficiency, particularly pronounced during parsing and deserialization tasks, with moderate yet notable advantages during serialization. Jackson showed distinct strengths in memory utilization, particularly during parsing, and competitive execution speeds for smaller datasets during serialization.

Gson generally performed less effectively relative to Jackson and KotlinX Serialization across all tested metrics, highlighting limitations due to its reflection-based approach. These results collectively indicate that KotlinX Serialization's compile-time code generation strategy significantly enhances performance, making it particularly suitable for performance-critical server applications handling large and complex JSON data. Jackson's reflection-based approach, while efficient in certain scenarios such as memory-constrained environments, exhibits scalability limitations under increasingly complex workloads.

Gson's overall performance disadvantages suggest it may be less suitable for scenarios requiring high computational efficiency or large-scale JSON data handling. In conclusion, this study provides clear, statistically validated empirical evidence highlighting the performance characteristics and trade-offs associated with three prominent JSON libraries in Kotlin server environments. These findings equip developers and architects with actionable insights, enabling informed library selection aligned with specific application requirements and performance objectives.

IV.2 DISCUSSION

IV.2.1 Interpretation Of Findings

The primary objective of this study was to systematically evaluate the comparative performance of three prominent JSON libraries—Jackson, Gson, and KotlinX Serialization—in server-side Kotlin environments. The analysis of experimental data consistently indicated significant performance differences among the tested libraries, each exhibiting distinctive advantages and limitations. KotlinX Serialization emerged prominently as the most efficient library overall, particularly excelling in execution time and CPU consumption during parsing and deserialization tasks. Jackson demonstrated notable strengths in memory utilization and serialization speed for smaller data sets. Conversely, Gson consistently underperformed relative to the other libraries in most performance metrics evaluated.

IV.2.2 Impact of Computational Strategies on Performance

The significant performance differences observed among these libraries can primarily be attributed to their underlying computational strategies. KotlinX Serialization's superior performance, especially regarding execution time and CPU efficiency, can be directly associated with its use of compile-time code generation through compiler plugins. This approach minimizes runtime overhead by avoiding reflection mechanisms commonly employed by Jackson and Gson. The elimination of runtime reflection substantially enhances computational efficiency and reduces resource demands, translating into consistently faster and more efficient data processing. Jackson and Gson rely heavily on runtime reflection, dynamically inspecting and processing data structures at runtime.

While reflection provides flexibility, enabling easy integration across various programming scenarios, it inherently introduces computational overhead and increased resource consumption. This reflection overhead is particularly pronounced in Gson, which consistently showed the lowest performance across execution time, memory usage, and CPU consumption metrics. Jackson, although similarly reliant on reflection, demonstrated superior memory efficiency and competitive execution speeds, particularly in serialization tasks involving smaller datasets. Jackson's performance optimization efforts, such as the use of specialized modules for Kotlin support, likely contribute to its improved performance relative to Gson [23].

IV.2.3 Memory Utilization Analysis

Memory usage results revealed another critical performance dimension. Jackson demonstrated superior memory efficiency during parsing tasks, consistently utilizing significantly less memory compared to Gson and KotlinX Serialization. This efficiency advantage suggests Jackson's reflection-based implementation, augmented by Kotlin-specific optimization modules, effectively manages memory resources during intermediate data processing.

Conversely, Gson's higher memory consumption is likely due to less optimized reflection handling, resulting in increased memory overhead during runtime. In serialization and deserialization tasks, KotlinX Serialization achieved substantial memory efficiency, significantly outperforming both Gson and Jackson. This enhanced efficiency likely results from KotlinX Serialization's compile-time code generation strategy, which streamlines object transformation processes and minimizes runtime memory allocations. This inherent advantage is particularly beneficial in resource-constrained or high-throughput server environments where memory optimization significantly influences overall performance and scalability [12].

IV.2.4 Cpu Consumption and Computational Load

As Analysis of CPU consumption further highlighted KotlinX Serialization's balanced and efficient approach. Consistently recording the lowest CPU usage across all tasks, KotlinX Serialization demonstrated effective computational optimization. The observed lower CPU usage is directly attributable to its compile-time generated serialization logic, reducing runtime complexity and computational demands. In contrast, Jackson and Gson's reflection-based methods inherently increase runtime processing complexity and CPU load, reflecting in consistently higher CPU consumption figures. However, the differences in CPU utilization between Jackson and Gson remained marginal, suggesting similar computational load characteristics inherent to reflection-based libraries.

IV.2.5 Practical Implications for Kotlin-Based Server Applications

From a practical perspective, the findings of this study offer actionable insights into library selection for server-side Kotlin applications. KotlinX Serialization's consistent superiority across multiple performance metrics—execution speed, CPU efficiency, and memory utilization in critical scenarios—makes it the preferred choice for high-performance applications, particularly those requiring rapid, large-scale JSON data processing. Its compile-time code generation strategy aligns closely with Kotlin's native features, ensuring optimal compatibility and performance.

Jackson remains an attractive alternative, particularly in scenarios prioritizing memory efficiency or involving moderate-sized datasets. Its reflection-based approach, supplemented by Kotlin-specific modules, provides sufficient performance for many server-side applications, balancing flexibility and resource usage. In contrast, Gson's lower overall performance metrics suggest limited suitability for demanding or resource-intensive server applications, where efficiency and computational load significantly impact application responsiveness and scalability.

IV.2.6 Study Limitations and Recommendations for Future Research

Footnotes Despite its comprehensive methodological approach, this study has certain limitations warranting consideration in interpreting results and guiding future research. Firstly, the scope was deliberately restricted to evaluating three widely adopted JSON libraries within Kotlin server environments. Future research could expand this scope to include additional libraries or different programming environments to offer broader comparative insights.

Secondly, the experiments were conducted under controlled laboratory conditions using specific hardware and software configurations. While this approach ensures methodological rigor and reproducibility, variations in hardware or deployment environments could influence performance outcomes. Therefore, future studies should consider conducting similar evaluations in diverse real-world deployment contexts to validate and extend these findings.

Additionally, the focus on JSON data with structured nesting and incremental complexity provides relevant insights; however, incorporating other data formats or processing scenarios could further enrich understanding of library performance characteristics. Further research could explore alternative serialization formats or real-world data processing tasks typical of industry-specific applications, thereby broadening the practical relevance and applicability of performance evaluations.

V. CONCLUSIONS

In conclusion, this discussion systematically examined the underlying factors influencing performance differences among Jackson, Gson, and KotlinX Serialization libraries, providing comprehensive interpretations of empirical findings. KotlinX Serialization's compile-time strategy consistently emerged as the most effective approach, delivering superior execution speed, memory efficiency, and CPU optimization suitable for high-performance Kotlin server applications. Jackson's memory efficiency and competitive performance in certain contexts provide valuable alternative options, while Gson's general underperformance highlights limitations inherent to reflection-based processing strategies. These insights enable informed decision-making among developers and system architects, ensuring optimal library selection aligned with specific server-side application requirements.

VI. AUTHOR'S CONTRIBUTION

Conceptualization: Achmad Arwan, Julia Nur Fajrina, Eriq Muhammad Adam Jonemaro.
Methodology: Achmad Arwan, Julia Nur Fajrina, Eriq Muhammad Adam Jonemaro.
Investigation: Achmad Arwan, Julia Nur Fajrina, Eriq Muhammad Adam Jonemaro.
Discussion of results: Achmad Arwan, Julia Nur Fajrina, Eriq Muhammad Adam Jonemaro.
Writing – Original Draft: Achmad Arwan, Julia Nur Fajrina, Eriq Muhammad Adam Jonemaro.
Writing – Review and Editing: Achmad Arwan, Julia Nur Fajrina, Eriq Muhammad Adam Jonemaro..
Resources: Achmad Arwan, Julia Nur Fajrina, Eriq Muhammad Adam Jonemaro.
Supervision: Achmad Arwan, Julia Nur Fajrina, Eriq Muhammad Adam Jonemaro.
Approval of the final text: Achmad Arwan, Julia Nur Fajrina, Eriq Muhammad Adam Jonemaro.

VII. REFERENCES

[1] A. Grebenkina, “Ten Years of Kotlin!,” 2021.

[2] JetBrains, “Kotlin - The State of Developer Ecosystem in 2023 Infographic,” 2025.

[3] JetBrains, “Kotlin Serialization Guide,” 2024.

[4] IETF, “RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format,” 2014.

[5] H. K. Dhalla, “A Performance Analysis of Native JSON Parsers in Java, Python, MS.NET Core, JavaScript, and PHP,” 16th International Conference on Network and Service Management, CNSM 2020, 2nd International Workshop on Analytics for Service and Application Management, AnServApp 2020 and 1st International Workshop on the Future Evolution of Internet Protocols, IPFutu, 2020, doi: 10.23919/CNSM50824.2020.9269101.

[6] Q. M. o‘g‘li Tohirov, D. A. Ayupova, and R. I. Zakirova, “Exploring Client-Side and Server-Side Architectures in Web Development: A Comprehensive Analysis,” 2024, doi: https://doi.org/10.5281/zenodo.10729101.

[7] M. Zoltu, T. Saloranta, and J. Minard, “Cache reflection results. · Issue #44,” 2016.

[8] J. Friesen, “Parsing and Creating JSON Objects with Gson,” in Java XML and JSON: Document Processing for Java SE, Apress, 2019, pp. 243–298. doi: https://doi.org/10.1007/978-1-4842-4330-5_9.

[9] JetBrains, “Kotlin multiplatform / multi-format serialization,” 2025.

[10] K. Maeda, “Performance evaluation of object serialization libraries in XML, JSON and binary formats,” 2012 2nd International Conference on Digital Information and Communication Technology and its Applications, DICTAP 2012, pp. 177–182, 2012, doi: 10.1109/DICTAP.2012.6215346.

[11] J. Vanura and P. Kriz, Perfomance evaluation of Java, JavaScript and PHP serialization libraries for XML, JSON and binary formats, vol. 10969 LNCS. Springer International Publishing, 2018. doi: 10.1007/978-3-319-94376-3_11.

[12] S. Aigner, “kotlinx.serialization 1.2 Released: High-Speed JSON Handling, Value Class Support, Overhauled Docs, and more,” 2021.

[13] JetBrains, “The State of Developer Ecosystem in 2023,” 2023.

[14] JetBrains, “Kotlin for Server Side,” 2024. [Online]. Available: https://kotlinlang.org/docs/server-overview.html

[15] Kotlin Documentation, “Serialization | Kotlin Documentation,” 2025. [Online]. Available: https://kotlinlang.org/docs/serialization.html

[16] Confluent, “What is Data Serialization? [Beginner’s Guide],” 2024. [Online]. Available: https://www.confluent.io/learn/data-serialization/

[17] B. Kitchenham, S. L. Pfleeger, and D. C. Hoaglin, “Statistical Significance Testing—A Panacea for Software Technology Experiments?,” Journal of Systems and Software, vol. 73, no. 3, pp. 485–501, 2002, doi: 10.1016/j.jss.2004.03.001.

[18] J. Gerrans and R. S. Sherratt, “Comparing XML and JSON Characteristics as Formats for Data Serialization Within Ultralow Power Embedded Systems,” IEEE Embed Syst Lett, vol. 16, no. 4, pp. 489–492, Dec. 2024, doi: 10.1109/LES.2024.3450576.

[19] H. K. Dhalla, “A Performance Analysis of Native JSON Parsers in Java, Python, MS.NET Core, JavaScript, and PHP,” in 2020 16th International Conference on Network and Service Management (CNSM), IEEE, 2020, pp. 1–5. doi: 10.23919/CNSM50824.2020.9269049.

[20] D. P. Proos and N. Carlsson, “Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV,” in 2020 IFIP Networking Conference (Networking), 2020, pp. 10–18.

[21] JetBrains, “Get started with Kotlin,” 2025.

[22] K. M. Ramachandran and C. P. Tsokos, “Chapter 12 - Nonparametric Tests,” in Mathematical Statistics with Applications in R (Second Edition), Academic Press, 2015, pp. 589–637. doi: https://doi.org/10.1016/B978-0-12-417113-8.00012-6.

[23] FasterXML, “Jackson Module Kotlin,” 2025.