INTERNATIONALE HOCHSCHULE

www.iu.de

IU DISCUSSION PAPERS

IT & Engineering

Serialization Trade-Offs in Distributed Systems

JOHANNES HABERLAH
KNUT LINKE

IU Internationale Hochschule

Main Campus: Erfurt

Juri-Gagarin-Ring 152

99084 Erfurt

Telefon: +49 421.166985.23

Fax: +49 2224.9605.115

Autorenkontakt:

Johannes Haberlah
ORCID-ID: 0009-0004-4774-7600 (Open Researcher und Contributer ID)

IU Internationale Hochschule – Campus Hamburg
Christoph-Probst-Weg 28
20251 Hamburg

E-Mail: johannes@haberlah.de

Prof. Dr. Knut Linke
ORCID-ID: 0000-0001-5088-5546 (Open Researcher und Contributer ID)

IU Internationale Hochschule - Campus Hannover

Schiffgraben 49-51

30175 Hannover

Telefon: +49-176-21100697

E-Mail: knutlinke@gmail.com

IU Discussion Papers, Reihe: IT & Engineering, Vol. 7, No. 3 (AUG 2026)

ISSN: 2750-073X

DOI: https://doi.org/10.56250/4143

Website: https://repository.iu.org

SERIALIZATION TRADE-OFFS IN DISTRIBUTED SYSTEMS

A Comparative Analysis of JSON, Protocol Buffers, and FlatBuffers Under Varying Width and Nesting Depth

Johannes Haberlah & Knut Linke

ABSTRACT:

This discussion paper compares three common serialization paradigms—JSON, Protocol Buffers, and FlatBuffers—with respect to latency and space efficiency. Although serialization is central to modern distributed systems, controlled, paradigm-level comparisons remain limited. To address this, the study benchmarks all three formats in Rust while systematically varying record width and nesting depth. Quantitative measurements are complemented by CPU profiling to identify dominant cost drivers.

Results show clear patterns: Protocol Buffers achieves the lowest serialization latency and smallest message sizes, FlatBuffers offers the fastest data access due to its zero-copy design, and JSON incurs the highest access costs and largest payloads. Each format therefore excels under different conditions: Protobuf in write-and bandwidth-sensitive scenarios, FlatBuffers in read-dominated contexts, and JSON where readability and flexibility matter.

KEYWORDS:

Serialization, Protocol Buffers, FlatBuffers, Zero-copy, Benchmarking

JEL: C80, C88, L86

AUTORS

Johannes Haberlah *is a software engineer in Zurich who began programming at the age of 12 and has since built extensive experience in software development. During his dual studies in Computer Science at IU, he worked as a Software Engineering Student at JAKALA, where he developed backend systems for clients such as ZDF and TUI Cruises. In 2025, he received the IU Spirit Award for an AI-based learning platform. During his studies, he also gained experience through a software engineering internship at a leading global technology company. After graduating, he joined the company as a full-time Software Engineer.*

Prof. Dr. Knut Linke *is a Professor of Computer Science at IU International University of Applied Sciences and a Professor of Digital Transformation, Information Technology, and AI at the University of Applied Sciences of the German Social Accident Insurance. His teaching and research focus primarily on artificial intelligence, digital transformation, and digital didactics. Another central focus of his work is to explain complex AI topics in an accessible way and to highlight their implications.*

Motivation, Research Questions and Hypotheses

Prior studies provide evidence that serialization format choice can substantially affect runtime and payload size, but they do not offer a controlled, paradigm-level comparison that isolates *structural complexity* as a driving factor. Müller et al. (2010) report strong size reductions for Protocol Buffers relative to XML; however, their focus is limited to a text-versus-binary comparison and relies on an outdated experimental environment (Müller et al., 2010). Proos and Carlsson (2020) compare Protocol Buffers and FlatBuffers in an Internet-of-Vehicles setting and find faster deserialization for FlatBuffers for larger messages, yet they neither include a textual baseline (e.g., JSON) nor systematically vary structural properties of the data (Proos & Carlsson, 2020).

As a result, a research gap remains: a systematic comparison of textual encoding, binary schema-based encoding, and zero-copy representations under controlled conditions, combining quantitative performance measurements with CPU-level root-cause analysis. This work addresses that gap by comparing JSON (textual), Protocol Buffers (binary), and FlatBuffers (zero-copy) with respect to serialization latency, access latency, throughput, and message size. To isolate structural complexity effects, the evaluation systematically varies record width (number of fields) and nesting depth (hierarchical structure), thereby probing how width and hierarchy influence end-to-end (de-)serialization costs.

To ensure a clear focus on paradigm-level differences rather than on implementation-specific variation within paradigms, the study selects one widely used representative per paradigm: JSON, Protocol Buffers, and FlatBuffers. Methodologically, the analysis uses a reproducible benchmark suite implemented in Rust and complements it with qualitative profiling to identify CPU hot spots and low-level bottlenecks specific to each format. Combining both perspectives enables not only quantification of observed performance differences, but also a causal explanation of why these differences occur.

Because prior work does not systematically control structural complexity, we explicitly vary record width and nesting depth to test whether format rankings remain stable across workloads. To address this gap, the study formulates explicit research questions that map directly to the key engineering trade-offs practitioners face when choosing an interchange format: serialization cost, access cost, and payload size. The corresponding hypotheses translate widely cited design expectations of the three paradigms into falsifiable statements, ensuring that dataset variation is interpreted against predefined claims rather than post-hoc reasoning. In addition, an exploratory question is included to connect macro-level performance differences to CPU-level mechanisms such as parsing overhead, heap allocations, and numeric conversion. This distinction between quantitative hypothesis testing and qualitative analysis supports causal interpretation rather than purely descriptive benchmarking.

MAIN RESEARCH QUESTION (RQ)

How do textual encoding (JSON), binary encoding (Protocol Buffers), and zero-copy formats (FlatBuffers) differ in serialization latency, access latency, and space efficiency under systematic variation of record width and nesting depth?

Sub-questions (SQ1-SQ4) and Hypotheses (H1-H5)

1. Which format achieves the lowest serialization latency?

Assumption: Binary formats allow streaming-friendly writes with sequential memory access and minimal structural overhead during encoding.

Hypotheses 1: Protocol Buffers has the lowest serialization latency among the evaluated formats.

2. Which format enables the fastest access to transferred data?

Assumption: Zero-copy approaches avoid conventional deserialization, reducing parsing overhead and heap allocations.

Hypotheses 2: FlatBuffers provides the lowest access latency.

Hypotheses 3: JSON exhibits the highest access latency.

3. How do the formats compare in message size?

Assumption: Binary formats without strict alignment padding use space more efficiently than textual formats or formats that enforce structural alignment.

Hypotheses 4: Protocol Buffers produces the smallest messages.

Hypotheses 5: JSON produces the largest messages.

4. Which low-level operations constitute the primary bottlenecks of each implementation?

This question is answered exploratorily via profiling, providing qualitative insight into parsing, allocation behavior, and CPU-intensive operations. No hypotheses are stated, as the goal is to complement the quantitative findings with causal explanations.

Theoretical Background

FUNDAMENTALS OF SERIALIZATION

Serialization is the transformation of structured, memory data into a linear byte sequence suitable for transmission or persistent storage. It enables interoperability between components that do not share memory and may be implemented in heterogeneous languages or architectures (Wolnikowski et al., 2021, p. 207). In distributed systems, this conversion is indispensable because messages must traverse process or network boundaries, and receivers must reconstruct semantically equivalent data structures through deserialization (Chawla, 2013, p. 230; Karandikar et al., 2021, p. 464).

Modern architectures amplify the importance of serialization: microservices and event-driven designs increase the number of (de-)serialization steps per end-to-end request, making serialization overhead a non-negligible contributor to latency and CPU utilization (Wolnikowski et al., 2021, pp. 206–207). Accordingly, both format selection and data-structure design can affect system performance.

Serialization formats can be categorized along three orthogonal dimensions (Maltsev, 2024; Proos & Carlsson, 2020; Sumaray & Makki, 2012):

1. Schema dependence: Schema-based formats rely on externally defined message structures (e.g., Protocol Buffers, FlatBuffers), while schema-less formats embed structure within the message (e.g., JSON).
2. Encoding type: Textual encodings use character-based, human-readable representations; binary encodings represent values in compact machine-oriented form.
3. Access model: Copy-based formats reconstruct in-memory objects during deserialization, whereas zero-copy formats enable direct access to fields within the serialized buffer.

TEXT-BASED SERIALIZATION: JSON

JSON is a lightweight, schema-less, human-readable text format with a minimal type system and grammar intentionally aligned with common programming data structures (Bray, 2014). Its simplicity and language independence contributed to broad adoption in web architectures, where it functions as a de-facto standard for client–server communication (Maltsev, 2024, p. 175; Vargas et al., 2019, p. 195).

As a self-describing format, JSON embeds field names directly within the message, enabling flexible evolution at the cost of syntactic redundancy and reduced type safety: applications must infer numeric or structural types at runtime (Bray, 2014, pp. 6–7). This design produces larger payloads and increases parsing work compared with schema-based binary formats.

JSON parsing typically involves sequential tokenization followed by syntactic validation, often implemented via finite-state automata (Li et al., 2017, p. 1118). Many implementations build an in-memory tree representation, which introduces fragmented allocations and pointer dereferences that impair cache locality (Peltenburg et al., 2021, p. 2). Research shows that JSON deserialization can dominate processing time in data-intensive workloads due to irregular control flow, limited parallelization, and runtime numeric conversion (Li et al., 2017, p. 1121; Maltsev, 2024, p. 178). Consequently, JSON is advantageous for human-centric use cases but comparatively inefficient in latency- or throughput-sensitive systems.

BINARY SCHEMA-BASED SERIALIZATION: PROTOCOL BUFFERS

Protocol Buffers (Protobuf) is a compact, schema-based binary format designed to support efficient message encoding in distributed systems (Varda, 2008). Message structures are defined in .proto files, from which language-specific code is generated. The schema enables static type checking, forward/backward compatibility, and a compact wire representation because field numbers—not field names—appear in the serialized output (Schwitzer, 2011; Karandikar et al., 2021).

The wire format consists of key–value pairs, where each key encodes the field number and wire type. Protobuf relies heavily on variable-length integers (Varints), enabling small values to be encoded in few bytes. For signed integers, ZigZag encoding maps negative values to small unsigned magnitudes to avoid inefficient multi-byte encodings. Length-delimited fields support nested messages and packed repeated values, providing efficient representation of hierarchical structures.

Protobuf's performance advantages stem from compact binary representation and sequential write patterns. However, Protobuf remains a copy-based format: deserialization requires reconstructing language-specific objects, which introduces allocation and copy overhead. Additionally, Protobuf is not human-readable and depends on the external schema for interpretation, which complicates ad-hoc debugging and browser-native use (Hemens, 2025, p. 32).

ZERO-COPY SERIALIZATION: FLATBUFFERS

Zero-copy formats address overhead inherent in copy-based deserialization by enabling direct access to serialized data without reconstructing an object graph. Unlike kernel-level zero-copy I/O, which reduces memory movement during network transmission, FlatBuffers implements application-level zero-copy parsing, avoiding full materialization of messages (Wolnikowski et al., 2021, pp. 206–207).

FlatBuffers represents data in a single contiguous buffer using relative offsets rather than absolute pointers, enabling random access to fields without sequential parsing. This internal layout (root offset, tables, and vtables) is documented in the FlatBuffers internals description (Google, n.d.). The format uses tables (flexible field presence) and vtables (per-object metadata describing field offsets). If a vtable entry is missing or zero, the accessor returns the schema-defined default value, supporting backward-compatible evolution. Scalars follow deterministic alignment rules; variable-length data (strings, vectors) is stored separately and referenced via offsets.

This design yields extremely low read latency dominated by offset arithmetic and direct memory loads. The trade-off is that writes are more complex than in streaming-oriented formats: the builder must construct vtables and offset structures in a consistent manner, and buffers are immutable once built. Modifications typically require reconstructing the buffer, reflecting FlatBuffers' focus on fast reads rather than flexible updates.

FlatBuffers is less widely adopted than JSON or Protobuf, but empirical evaluations demonstrate lower memory usage and reduced deserialization time, particularly in resource-constrained or latency-sensitive scenarios (Proos & Carlsson, 2020; Maltsev, 2024, pp. 176–177).

Methodology

This study follows a sequential two-stage research design. First, quantitative benchmarking is used to measure statistically robust differences in runtime behavior across formats (addressing SQ1–SQ3). Second, CPU profiling is used to support interpretation of these differences by identifying format-specific cost drivers and explaining anomalies (addressing SQ4).

This pattern, quantitative measurement followed by profiling-based root-cause analysis, is common in performance research. For example, Parimi et al. (2019) benchmark Protobuf deserialization before performing function-level profiling with perf. Similarly, Karandikar et al. (2021) quantify infrastructure-level Protobuf overheads and subsequently identify dominant micro-operations through profiling.

Given hypotheses H1–H5, quantitative benchmarking forms the primary methodological component, while profiling provides an explanatory layer that strengthens causal interpretation without claiming causal proof on its own.

BENCHMARK ENVIRONMENT IMPLEMENTATION

The benchmark suite was implemented in Rust, selected for its low-level execution model and absence of nondeterministic managed-runtime behavior. Because Rust compiles directly to machine code, it reduces, though does not eliminate, runtime-environment confounding effects commonly observed in languages with garbage collection or JIT compilation (Lunnikivi, 2020, p. 10). Rust's predictable memory model also integrates well with Linux profiling tools such as perf and valgrind, which is necessary for the qualitative stage.

For benchmarking, the study uses criterion.rs, a de-facto standard in the Rust ecosystem (Cai, 2023; Färnstrand, 2015; Lunnikivi, 2020; Motshegwe, 2025; van Seventer, 2025). Criterion provides statistical rigor beyond manual timing, including automated warmup phases and noise reduction (Färnstrand, 2015, p. 31). It employs adaptive sampling to determine appropriate iteration counts and uses linear regression to separate constant measurement overhead from per-operation cost (Junge, 2022, pp. 11–13).

For profiling, Linux perf is used as a sampling-based profiler capable of attributing CPU time to code regions with low overhead (Van der Post, 2024, pp. 198–199). Flamegraphs generated via cargo-flamegraph visualize stack frequency distributions and highlight dominant execution paths. As Criterion's adaptive control flow is unsuitable for profiling, separate runner programs repeatedly execute fixed (de)serialization operations. For FlatBuffers serialization, the benchmark instantiates a new builder per operation (i.e., no cross-iteration builder reuse), so measured write costs include builder allocation and buffer growth behavior.

To support reproducibility, the complete benchmark suite (schemas, runners, and analysis artifacts) is publicly available (Haberlah, 2026).

DEFINITION OF TEST DATASETS

To study structural effects in a controlled manner, datasets vary along two dimensions:

- Width: number of fields per message
- Depth: nesting level of structural containment

These dimensions influence (de)serialization cost through field count and hierarchical structure (Zeng et al., 2023; Bittl, 2015; Maltsev, 2024).

Tab. 1: Dataset characteristics width, depth, and type mix (own work)

Dataset	Variant	Width / Payload fields	Nesting depth	Type mix (high-level)
Dataset 1	small, shallow	5 fields	1	u64, string×2, bool, repeated string
Dataset 2	small, deep	5 fields (at leaf) + level identifiers	6	leaf: u64, string×2, bool, repeated string; plus 5× level_id (string) across nesting chain
Dataset 3	large, shallow	28 fields	1	integers (u32/u64/i64), float (f64), bool, string, repeated string vectors
Dataset 4	large, deep	LargeFlat payload (28 fields) + nested chain	6	LargeFlat type mix + 5× level_id (string) across nesting chain

Four datasets systematically combine high/low width with high/low depth. Table 1 summarizes the structural characteristics of the four benchmark datasets. “Width” refers to the number of fields in the primary record, while “depth” denotes the number of nested levels that must be traversed to access the payload.

A heterogeneous set of data types is used to avoid overfitting results to a single primitive type, although specific type distributions are documented only in the benchmark implementation. This approach aligns with dataset-construction strategies in prior serialization-performance research (Cao, 2022, p. 7; Jang, 2020, p. 3).

QUANTITATIVE RUNTIME ANALYSIS

The benchmark measures three metrics across all datasets and formats:

- Latency: mean time per (de-)serialization operation, estimated as the slope of a regression model. For FlatBuffers, latency refers to full data access since no conventional deserialization occurs.
- Throughput: number of fully processed elements per second (Melem/s), consistent with definitions used in the result tables.
- Message size: serialized payload length in bytes.

Latency and throughput directly operationalize SQ1 and SQ2, while message size operationalizes SQ3.

CPU and memory utilization metrics are intentionally omitted because latency/throughput and message size already capture the primary performance characteristics relevant to interchange formats, and raw CPU/RAM readings would be strongly dependent on language- and run-time-specific behavior.

Individual (de-)serialization operations often complete within nanoseconds or microseconds, making direct per-operation measurement unreliable due to timer overhead and noise (Junge, 2022, pp. 11–13).

Criterion addresses this by timing blocks of operations. Adaptive sampling selects block sizes long enough for stable timing, and linear regression isolates fixed overhead from per-operation cost, improving estimation accuracy and variance characteristics. This method is widely accepted in micro benchmarking literature.

QUALITATIVE PROFILING ANALYSIS

The qualitative phase complements quantitative results by identifying CPU hotspots associated with (de)serialization. perf performs statistical sampling of the current instruction pointer; with sufficient samples, the relative frequency approximates each function’s contribution to CPU time (subject to sampling granularity and stack-unwinding reliability).

Flamegraphs depict call stacks proportionally to their sampling frequency, enabling intuitive identification of dominant routines. Because Criterion’s control logic interferes with sampling, dedicated runner programs repeatedly invoke specific functions for profiling.

TEST ENVIRONMENT SPECIFICATION

All benchmarks were executed on an isolated AWS EC2 c6i.2xlarge instance to reduce background interference. The configuration was:

Hardware

- CPU: Intel Xeon “Ice Lake” (x86_64)
- RAM: 16 GB

Operating system

- Debian 12.13 (bookworm), kernel 6.1.0–18-cloud-amd64
- perf: linux-perf 6.1.0

Toolchain

- Rust: rustc 1.90.0
- Benchmark: criterion 0.8.1

Serialization libraries

- JSON: serde_json 1.0.149
- Protobuf: prost 0.14.3
- FlatBuffers: flatbuffers 25.12.19

AWS c6i machines provide stable performance characteristics for microbenchmarks, although they do not eliminate all virtualization effects.

Results

Reminder. As defined in the methodology section, “small” and “large” denote low vs. high record width, while “shallow” and “deep” denote low vs. high nesting depth. The four datasets combine these dimensions systematically.

SERIALIZATION RUNTIME

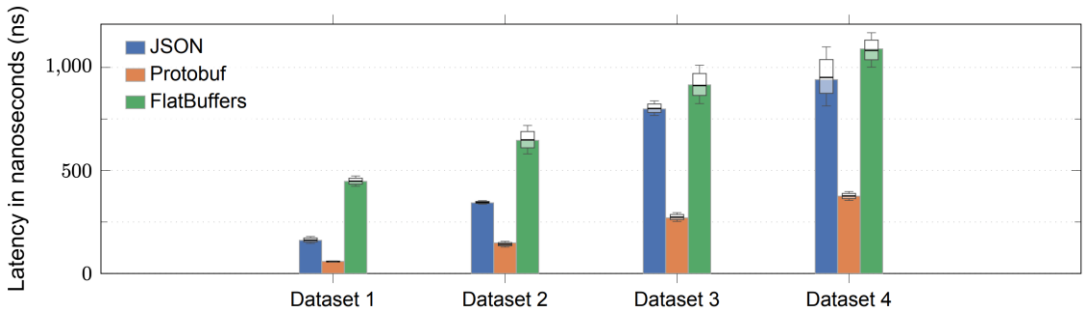
To answer research question SQ1 (which technology achieves the lowest serialization latency), Figure 1 reports the measured serialization latencies across all datasets, and Table 1 provides the corresponding numeric values. Latencies represent the slope of a linear regression model following the measurement procedure described in the previous section; one element corresponds to the complete serialization of a dataset instance.

Across all datasets, the ranking is consistent: Protobuf achieves the lowest serialization latency, JSON occupies the middle position, and FlatBuffers has the highest latency. For instance, in Dataset 1, Protobuf requires 58.58 ns, compared to 160.70 ns for JSON and 446.75 ns for FlatBuffers. This ordering holds even as structural complexity increases (Dataset 4: 375.25 ns vs. 940.50 ns vs. 1089.91 ns). Confidence intervals do not overlap, indicating robust differences.

Protobuf outperforms JSON by factors between 2.31× and 2.96×, depending on dataset, while FlatBuffers is between 1.15× and 2.78× slower than JSON. Throughput (Melem/s) behaves inversely to latency, with Protobuf achieving the highest throughput and FlatBuffers the lowest.

These results consistently support H1: Protobuf yields the lowest serialization latency across all examined conditions.

Fig. 1: Serialization latency by technology and dataset



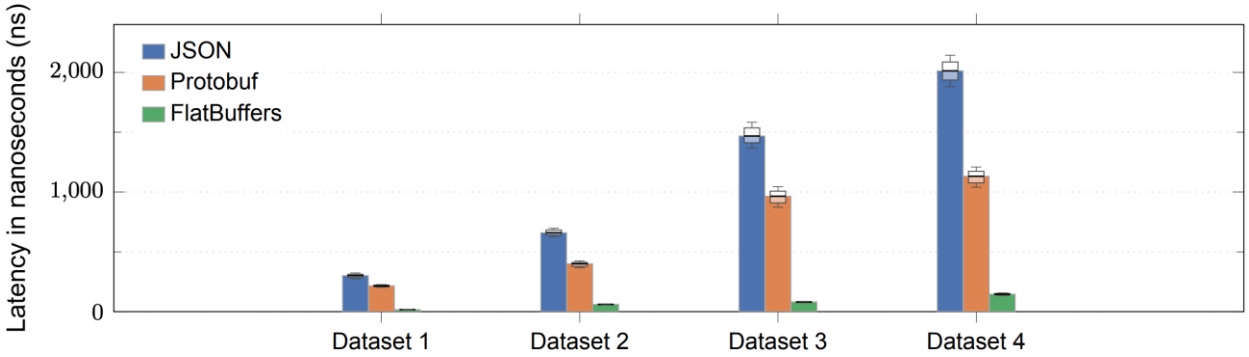
Tab. 2: Serialization latency per element (ns) across four structurally varied datasets (own work)

Benchmark	JSON	Protobuf	FlatBuffers
Dataset 1 (small, shallow)	160.70	58.58	446.75
Dataset 2 (small, deep)	342.85	148.29	645.91
Dataset 3 (large, shallow)	798.26	269.84	915.40
Dataset 4 (large, deep)	940.50	375.25	1089.91

DESERIALIZATION RUNTIME (DATA ACCESS)

To answer SQ2 (which technology provides the lowest latency for accessing transmitted data), Figure 2 shows deserialization/access latencies, and Table 2 provides numeric results. For FlatBuffers, which does not perform deserialization in the conventional sense, the benchmark measures a full data traversal covering all fields.

Fig. 2: Data-access latency by technology and dataset



The ordering reverses compared to serialization: FlatBuffers is by far the fastest for data access, followed by Protobuf, with JSON as the slowest technology. For Dataset 1, FlatBuffers requires 16.56 ns, compared to 215.31 ns (Protobuf) and 301.16 ns (JSON). The same pattern persists in Dataset 4 (145.68 ns vs. 1131.99 ns vs. 2012.78 ns). Again, confidence intervals do not overlap.

FlatBuffers accelerates access relative to Protobuf by 6.61×–13.00× and relative to JSON by 10.89×–18.18×. Variability is lowest for FlatBuffers due to its deterministic, allocation-free access model.

Tab. 3: Data-access latency per element (ns) across four structurally varied datasets (own work)

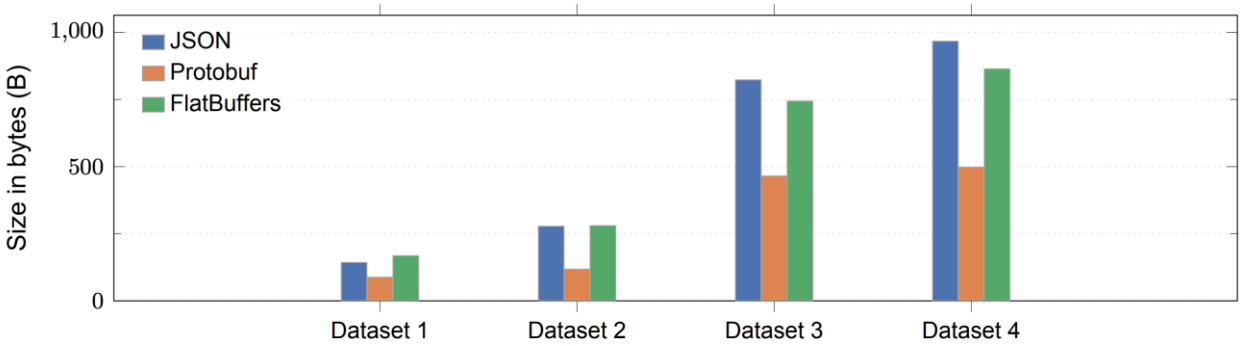
Benchmark	JSON	Protobuf	FlatBuffers
Dataset 1 (small, shallow)	301.16	215.31	16.56
Dataset 2 (small, deep)	659.48	399.97	60.55
Dataset 3 (large, shallow)	1468.17	964.64	80.83
Dataset 4 (large, deep)	2012.78	1131.99	145.68

These findings support H2 (FlatBuffers lowest latency) and H3 (JSON highest latency).

MESSAGE SIZE

Research question SQ3 concerns the relative compactness of serialized messages. Figure 3 visualizes message sizes, and Table 3 provides numeric values.

Fig. 3: Serialized message size by technology and dataset



Tab. 4: Serialized message size (bytes) for all datasets (own work)

Benchmark	JSON	Protobuf	FlatBuffers
Dataset 1 (small, shallow)	143	88	168
Dataset 2 (small, deep)	278	118	280
Dataset 3 (large, shallow)	823	465	744
Dataset 4 (large, deep)	967	498	864

Across all datasets, Protobuf produces the smallest payloads. Relative to JSON, Protobuf achieves reductions of 38.5%–57.6%. The relative ordering between JSON and FlatBuffers is dataset dependent: Notably, the crossover is marginal in Dataset 2 (FlatBuffers 280 B vs. JSON 278 B), indicating that the boundary for “small payloads” is narrow under this workload. FlatBuffers is larger for small datasets but becomes smaller for larger datasets, reflecting the fact that vtable/offset metadata dominates small messages but amortizes as payloads grow.

Thus, H4 is fully supported (Protobuf smallest messages), while H5 is only partially supported: JSON is largest for large datasets, but FlatBuffers exceeds JSON for small payloads.

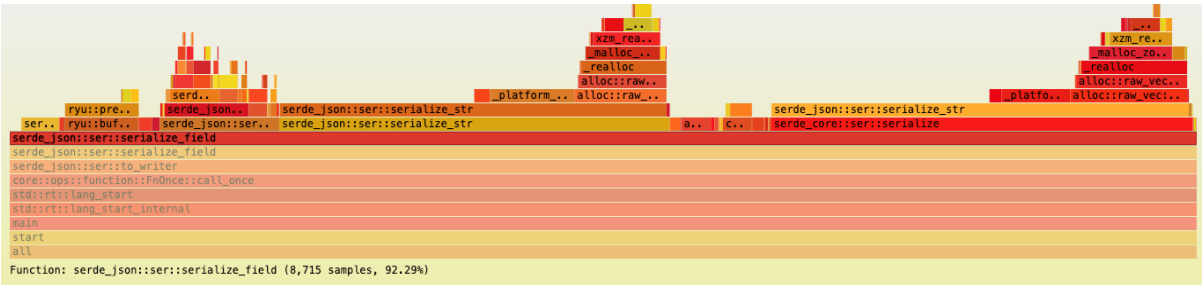
PROFILING RESULTS (CPU HOTSPOTS)

To address SQ4, CPU sampling profiles were examined to identify root causes behind the measured performance differences. Because sampling-based profiling provides statistical estimates, reported percentages should be interpreted as approximate and sensitive to profiling duration and sampling configuration.

Across formats, profiling highlights the following dominant mechanisms:

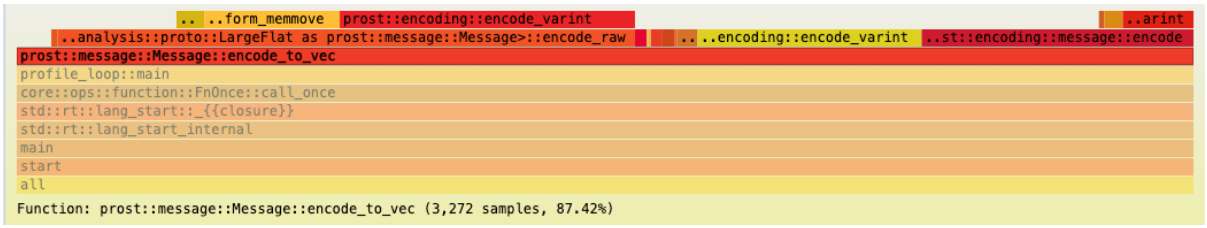
JSON: Hotspots originate from textual numeric formatting and string emission. Functions such as floating-point formatting (format64) and key serialization dominate processing, alongside buffer reallocation. These reflect JSON's structural redundancy and conversion overhead.

Fig. 4: Flamegraph of JSON serialization showing dominant formatting and string-emission hotspots



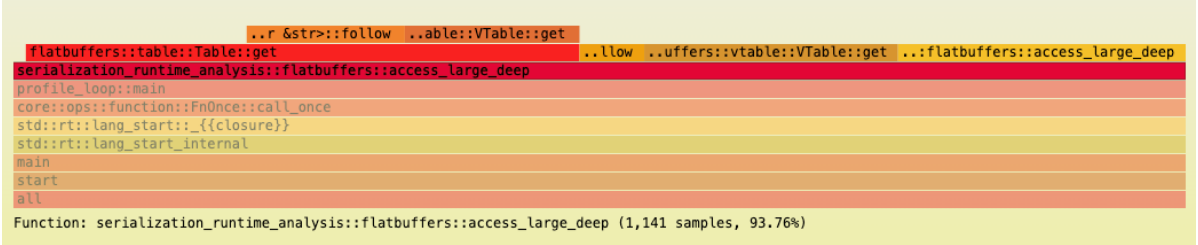
Protobuf: Protobuf's hotspots are tightly concentrated in varint encoding and field wise byte emission (encode_varint, encode_raw). Memory operations such as memmove and dealloc contribute secondary overhead. The profiling pattern aligns with Protobuf's binary, schema-based design.

Fig. 5: Flamegraph of Protobuf serialization highlighting varint and field-write operations



FlatBuffers: FlatBuffers exhibits near exclusive activity in offset resolution and table/vtable lookup, confirming its zero-copy nature. No heap allocations appear. The performance is dominated by constant time offset operations (Table::get, VTable::get), consistent with the extremely low data-access latencies observed.

Fig. 6: Flamegraph of FlatBuffers data access dominated by table/vtable offset resolution



These profiling-based interpretations explain the quantitative findings of the last Sections and answer SQ4 by identifying format-specific root causes.

Discussion

SERIALIZATION RUNTIME

Across all four datasets, the serialization benchmarks show a stable and unambiguous ranking: Protobuf is consistently the fastest, followed by JSON, while FlatBuffers exhibits the highest serialization latency. The 95% confidence intervals do not overlap for any dataset, demonstrating statistical robustness.

Quantitatively, the differences are meaningful. For Dataset 1, Protobuf's mean serialization time is 58.58 ns, compared to 160.70 ns for JSON and 446.75 ns for FlatBuffers. The same ordering holds under increased structural complexity (Dataset 4: 375.25 ns vs. 940.50 ns vs. 1089.91 ns). Throughput follows the inverse pattern, with Protobuf achieving the highest values (up to 17.07 Melem/s) and FlatBuffers the lowest.

The strong performance of Protobuf aligns with its design: the binary wire format avoids textual conversion and does not repeat structural metadata. Profiling supports this explanation. The dominant routines (encode_varint, encode_raw) together account for more than 80% of CPU samples, reflecting Protobuf's reliance on byte-level encoding and sequential writes. This concentration of work is structural, not pathological—serialization effort is intentionally focused on compact integer encoding and direct buffer emission.

JSON's intermediate performance matches expectations for a textual, schema-less format. Numeric values must be converted to decimal strings, which is visible in profiling traces (format64, write_mantissa_long). Additional overhead stems from key emission and dynamic buffer growth (serialize_key, serialize_str, memmove, dealloc). These operations reflect JSON's inherent syntactic redundancy and lack of type specialization.

FlatBuffers has the highest serialization cost due to its architectural trade-offs. The zero-copy access model requires constructing vtables, calculating field offsets, and satisfying alignment requirements. These steps introduce overhead that is more noticeable in the smaller datasets, where metadata occupies a larger relative share. As messages scale, the cost becomes better amortized, but it never reaches the efficiency of Protobuf's streaming style writes.

Across datasets, Protobuf exceeds JSON by 2.31×–2.96×, while FlatBuffers is 1.15×–2.78× slower than JSON. An exception appears in Dataset 2, where Protobuf shows unusually high variability (SD = 92 ns). This likely stems from system-level noise or microarchitectural sensitivity under deep nesting rather than from the format itself. The order remains unchanged.

Overall, the results support H1: Protobuf achieves the lowest serialization latency across all datasets. JSON's overhead derives from text conversion and structural redundancy, while FlatBuffers trades serialization cost for superior read performance.

DESERIALIZATION RUNTIME (DATA ACCESS)

The access-time benchmarks show a clear and consistent ranking across all datasets: FlatBuffers is by far the fastest, followed by Protobuf, while JSON exhibits the highest latency. Confidence intervals again do not overlap.

FlatBuffers' access times are exceptionally low: 16.56 ns (Dataset 1) and 145.68 ns (Dataset 4); producing speedups of 10.89×–18.18× over JSON and 6.61×–13.00× over Protobuf. This reflects FlatBuffers' architectural goal: avoid conventional deserialization entirely and operate directly on the received memory buffer.

Profiling confirms this: more than 93% of CPU samples fall into direct-access operations. The dominant routines, Table::get and VTable::get, correspond to offset resolution and vtable lookup: the core of the zero-copy paradigm. No heap allocations appear in the traces, supporting the conclusion that FlatBuffers achieves deterministic O(1) field access without object construction.

Protobuf occupies the middle position. As a copy-based format, it must parse binary data, decode varints, and materialize typed message structures. This introduces work absent in FlatBuffers but still far less than JSON's dynamic, schema-less parsing. Speedups relative to JSON range from 1.40× to 1.78×.

JSON is consistently slowest. Parsing requires tokenization, numeric conversion, and constructing a DOM-like structure, which appears in the traces through frequent allocation-related routines. Its schema-less nature further triggers runtime type inference, adding processing overhead.

These results support H2 and H3:

- H2: FlatBuffers achieves the lowest access latency.
- H3: JSON exhibits the highest access latency.

The ranking matches architectural expectations: zero-copy (FlatBuffers) < binary copy-based (Protobuf) < textual copy-based (JSON).

MESSAGE SIZE

The message-size results show that Protobuf produces the smallest serialized payload across all datasets, reducing JSON's size by 38.5%–57.6%. This reflects Protobuf's compact wire format, which encodes field identifiers as numeric tags and represents integers using varints.

The relative ordering between JSON and FlatBuffers varies with dataset size. For small datasets, FlatBuffers is larger than JSON because vtables and alignment padding dominate the relatively small payload. For large datasets, this overhead becomes negligible, and FlatBuffers overtakes JSON.

JSON produces the largest messages for the large datasets, driven by the repetition of structural metadata (keys, punctuation) and text-based numeric encoding. The reductions relative to JSON align with findings from Maltsev & Muliarevych (2024).

Thus, H4 is fully supported (Protobuf smallest messages).

H5 is partially supported: JSON is largest in the large datasets, but FlatBuffers is largest in the small ones.

OVERALL RESEARCH QUESTION AND PRACTICAL RECOMMENDATIONS

The research sought to understand how textual, binary, and zero-copy serialization paradigms differ in runtime and space efficiency under structural variation. Across all metrics, the results display a coherent and architecture-consistent pattern:

- Protobuf is optimal for write-dominated workloads due to its compact representation and fast serialization.
- FlatBuffers is optimal for read-dominated workloads requiring rapid access and minimal overhead.
- JSON remains suitable where human readability and maximal ecosystem compatibility outweigh raw performance considerations.

Hypotheses H1–H4 are supported; H5 receives partial support due to FlatBuffers' small-message metadata overhead.

The results underscore that serialization is not merely an implementation detail but a substantive architectural choice with significant implications for latency, throughput, and payload size. In practice, the choice between Protobuf and FlatBuffers depends strongly on the read-to-write ratio. FlatBuffers' higher write cost can be amortized when serialized messages are accessed multiple times (e.g., cached buffers, repeated traversal, memory-mapped data), whereas Protobuf is preferable when messages are typically written once and consumed once.

LIMITATIONS AND DIRECTIONS FOR FUTURE RESEARCH

Although the evaluation provides clear evidence for the performance differences among JSON, Protocol Buffers, and FlatBuffers, several limitations restrict the generalizability of the findings.

Limitations:

1. All benchmarks were implemented in Rust using specific libraries (serde_json, prost, flatbuffers). This ensures deterministic execution without garbage-collection effects but means the results primarily reflect these implementations. Other languages with JIT compilation or managed runtimes may show different absolute and relative performance.
2. The datasets used in this study vary width and depth systematically but remain synthetic and relatively small. They do not cover real-world heterogeneous schemas, multi-megabyte messages, or streaming workloads. In such scenarios, especially with repeated keys or deeply nested structures, the relative behavior of the formats may change.
3. Measurements were performed on a single Intel Xeon (x86_64) platform. Different architectures, especially ARM or specialized accelerators, may influence caching behavior, prediction accuracy, and memory bandwidth, potentially affecting performance.
4. Profiling provides qualitative insight but limited microarchitectural detail. The flamegraph fragments available in the context are lower resolution and partially truncated, preventing a

full analysis of cache behavior, pipeline stalls, or branch mispredictions. Profiling therefore supports, but cannot conclusively prove, the causal mechanisms inferred from theory.

5. The evaluation isolates raw (de)serialization. In real systems, serialization interacts with network protocols, compression, batching, and schema evolution, which may amplify or reduce the differences observed here.

Directions for Future Research:

1. Compression sensitivity: Evaluate how much of the observed message-size differences persist under standard compression (e.g., gzip or zstd), which is commonly enabled in production deployments.
2. Cross-language replication: Repeating the benchmarks in Java, Go, Python, or C++ could help disentangle paradigm-level effects from language-runtime effects.
3. Intra-paradigm comparisons: Comparing alternative libraries—e.g., simdjson vs. serde_json, multiple Protobuf runtimes, or FlatBuffers vs. Cap'n Proto—would clarify how much performance is driven by library engineering rather than paradigm differences.
4. Evaluation with larger and real-world schemas: Testing multi-megabyte payloads or production-grade data models (telemetry, IoT, gaming) would improve ecological validity and stress-test zero-copy designs.
5. Microarchitectural profiling: Tools such as perf stat, Intel VTune, or Cachegrind could reveal cache locality, TLB behavior, and instruction-level bottlenecks, complementing the high-level hotspot results.
6. End-to-end system studies: Embedding the formats into real microservice or streaming pipelines would allow studying the interplay between serialization, networking, batching, and service latency.

Conclusion

Recent analyses indicate that a noticeable share of CPU cycles in large-scale systems such as Google's infrastructure is spent on serialization-related work (Karandikar et al., 2021). This motivated the present study, which compared three widely used serialization paradigms, JSON as a textual format, Protocol Buffers as a binary schema-based format, and FlatBuffers as a zero-copy approach, regarding their latency and space efficiency.

Using a Rust-based benchmark suite, complemented by profiling to better understand the underlying causes, the study observed clear and consistent tendencies. Protocol Buffers generally showed the lowest serialization latencies and the smallest message sizes. FlatBuffers, in turn, offered by far the fastest access times, benefiting from its ability to avoid conventional deserialization. JSON, as expected for a textual and schema-less format, showed higher access costs and larger message sizes in most scenarios.

These results do not point to a universally superior format but rather highlight different strengths depending on the workload. Protocol Buffers is a good fit for scenarios where efficient writing and compact messages are important. FlatBuffers become attractive when rapid data access is essential.

JSON continues to be valuable were readability, interoperability, or tooling support matter more than raw performance.

At the same time, the findings should be interpreted considering several limitations. The benchmarks were implemented in Rust and rely on specific libraries, which may behave differently in other languages or runtime environments. The datasets, although systematically varied, do not capture all the complexities of real-world schemas. Hardware characteristics and the sampling-based nature of the profiling also influence the results.

These limitations open several directions for further work. Replicating the benchmarks in other languages, comparing alternative libraries within the same paradigm, or considering much larger and more diverse schemas could broaden the picture. More fine-grained microarchitectural profiling or evaluations in end-to-end distributed systems would further clarify how these formats behave under realistic conditions.

Overall, the study underlines that serialization format choice is not a minor detail but can have noticeable impact on system performance and that the most suitable format depends strongly on the intended use case.

References:

Amann, M., Baumann, J., & Koch, M. (2022). *Rust: Konzepte und Praxis für die sichere Anwendungsentwicklung* (1. Aufl.). dpunkt.verlag.

Bittl, S., Gonzalez, A. A., Spahn, M., Heidrich, W. A., & Esk, F. (2015). *Performance Comparison of Data Serialization Schemes for ETSI ITS Car-to-X Communication Systems.*

Bray, T. (2014). *The JavaScript Object Notation (JSON) Data Interchange Format (RFC7158).* https://doi.org/10.17487/rfc7158

Cai, Y., Singh, P., Lin, Z., Bosamiya, J., Gancher, J., Surbatovich, M., & Parno, B. (2023). *Vest: Verified, Secure, High-Performance Parsing and Serialization for Rust.*

Cao, S., Di Girolamo, S., & Hoefler, T. (2022). Accelerating data serialization/deserialization protocols with in-network compute. *IEEE/ACM International Workshop on Exascale MPI*, 22–30. https://doi.org/10.1109/ExaMPI56604.2022.00008

Chawla, R. (2013). Object serialization formats and techniques: A review.

Färnstrand, L. (2015). *Creating the ForkJoin framework.*

Google. (n.d.). *FlatBuffers Internals.* https://flatbuffers.dev/internals/

Haberlah, J. (2026). *serialization-benchmark-report* [GitHub repository]. GitHub. Retrieved August 5, 2026, from https://github.com/johanneshaberlah/serialization-benchmark-report/tree/main

Hemens, R. (2025). *Automatic Instrumentation With OpenTelemetry for Software Visualization.*

Jackson, S., Cummings, N., & Khan, S. (2024). Streaming technologies and serialization protocols: Empirical performance analysis. *IEEE Access, 12*, 158025–158039. https://doi.org/10.1109/ACCESS.2024.3486054

Jang, J., Jung, S. J., Jeong, S., Heo, J., Shin, H., Ham, T. J., & Lee, J. W. (2020). A specialized architecture for object serialization with applications to big data analytics. *ISCA*, 322–334. https://doi.org/10.1109/ISCA45697.2020.00036

Junge, A. (2022). *Reactive Benchmarking – Benchmarking in the Futhark compiler.*

Karandikar, S., Leary, C., Kennelly, C., Zhao, J., Parimi, D., Nikolic, B., Asanovic, K., & Ranganathan, P. (2021). A hardware accelerator for Protocol Buffers. *MICRO-54*, 462–478. https://doi.org/10.1145/3466752.3480051

Koparkar, C. (2022). Efficient data representation using FlatBuffers. *XRDS, 29*(2), 50–51. https://doi.org/10.1145/3571304

Li, Y., Katsipoulakis, N. R., Chandramouli, B., Goldstein, J., & Kossmann, D. (2017). Mison: A fast JSON parser for data analytics. *VLDB, 10*(10), 1118–1129.

Lunnikivi, H., Jylkkä, K., & Hämäläinen, T. (2020). Transpiling Python to Rust for optimized performance. In *Embedded Computer Systems* (pp. 127–138). Springer.

Maltsev, E., & Muliarevych, O. (2024). Beyond JSON: Evaluating serialization formats for space-efficient communication. *Advances in Cyber-Physical Systems, 9*(1), 9–15.

Maltsev, E., Muliarevych, O., & Razzaque, A. (2024). Classifying serialization formats for inter-service communication in distributed systems. *Advances in Cyber-Physical Systems, 9*(2), 175–180.

Möller, J., Weißberg, F., Pirch, L., Eisenhofer, T., & Rieck, K. (2024). Cross-language differential testing of JSON parsers. *ACM Asia CCS*, 1117–1127.

Motshegwe, B. (2025). *A Comparative Study of C++ and Rust in Developing High-Performance, Low-Latency Systems.*

Müller, J., Lorenz, M., Geller, F., Zeier, A., & Plattner, H. (2010). Assessment of communication protocols in the EPC network. *ICIEEM*, 404–409.

Parimi, D., Zhao, W. J., & Zhao, J. (2019). *Datacenter tax cuts: Improving WSC efficiency through Protocol Buffer acceleration.*

Peltenburg, J., Hadnagy, A., Brobbel, M., Morrow, R., & Al-Ars, Z. (2021). Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators. *ICFPT*, 1–9.

Proos, D. P., & Carlsson, N. (2020). Performance comparison of messaging protocols and serialization formats for digital twins. *IFIP Networking*, 10–18.

Protocol Buffers. (n.d.). *Encoding*. Retrieved August 5, 2026, from https://protobuf.dev/programming-guides/encoding/

Protocol Buffers. (2024). *Editions overview*. Retrieved August 5, 2026, from https://protobuf.dev/editions/overview/

Raghavan, D., Levis, P., Zaharia, M., & Zhang, I. (2021). Breakfast of champions: Towards zero-copy serialization with NIC scatter-gather. *HotOS*, 199–205.

Schwitzer, W., & Popa, V. (2011). Using Protocol Buffers for resource-constrained distributed embedded systems.

Sumaray, A., & Makki, S. K. (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. *ICUIMC*, 1–6.

Van der Post, H. (2024). *Data Science with Rust.*

van Oortmerssen, W. (2014). FlatBuffers: A memory-efficient serialization library. https://android-developers.googleblog.com

van Seventer, R. (2025). *Scalable Structural Code Diffs.*

Varda, K. (2008). Protocol Buffers: Google's data interchange format. https://opensource.googleblog.com

Vargas, S., Goel, U., Steiner, M., & Balasubramanian, A. (2019). Characterizing JSON traffic patterns on a CDN. *IMC*, 195–201.

Wolnikowski, A., Ibanez, S., Stone, J., Kim, C., Manohar, R., & Soulé, R. (2021). Zerializer: Towards zero-copy.

Zeng, X., Hui, Y., Shen, J., Pavlo, A., McKinney, W., & Zhang, H. (2023). An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment*, *17*(2), 148–161. https://doi.org/10.14778/3626292.3626298