Comparative Evaluation of NakedBytes, FlatBuffers, and Protocol Buffers for Efficient Data Serialization, Deserialization and Storage

Busoye Tolulope Matthew

December 19, 2025

Abstract

1 Introduction

2 Related Work

3 Methodology

This paper present a comparative analysis on the three serialization framework with the aim to give a holistic analysis of them. The analysis focuses on the following major parameters:

- Resulting Binary: This deals with the resulting storage requirement of the serialized binary output.
- Size: This includes program size, and static ram usage.
- Computation time: This includes the time required for serialization and Deserialization of the data into usage format int he native programming languages.
- Memory Usage: This includes the stack and heap allocation requirement during serialization and Deserialization.
- Compatibility: This includes forward and backward compatibility.

To experiment and analysis were curated to be able to determine the above parameters analytical while removing noise.

The experiment was performed on the following data struct:

The experiment carried out are then divided:

- A simple code that contains utility code and functions and test are carried out to determine the default overhead added.
- A Deserialization alone.
- A Deserialization with one field access.

- A deserialization with all fields access.
- A deserialization with nested fields.
- A serialization alone.
- A serialization with one field access.
- A serialization with all fields access.
- A serialization with nested fields.
- A serialization and Deserialization alone.
- A serialization and Deserialization with one field access.
- A serialization and deserialization with all fields access.
- A serialization and deserialization with nested fields.

These experiment are both performed on the Dell Inc. Precision 5530 Intel® Core™ i9-8950HK × 12 Ubuntu 24.04.3 LTS 64-bit pc and rp2350 Dual Arm Cortex-M33 (150MHz) 520 KB 32 bit.

4 Methodology

This paper presents a comparative experimental evaluation of three serialization frameworks—NakedBytes, FlatBuffers, and Protocol Buffers—with the objective of providing a holistic and practically relevant analysis of their behavior across constrained and general-purpose systems. The evaluation is designed to capture not only serialized size, but also runtime, memory usage, and compatibility characteristics that are critical in energy-sector and embedded deployments.

4.1 Evaluation Parameters

The analysis focuses on the following primary parameters:

- **Resulting Binary Size**: The size of the serialized output buffer produced by each framework for equivalent data structures. This metric captures the efficiency of the on-wire or stored representation, including all internal metadata.
- **Code Size and Static Memory Usage**: This includes the compiled program size (flash/ROM footprint) and static RAM usage introduced by the serialization framework and generated code.
- **Computation Time**: The execution time required for serialization and deserialization operations, measured separately. This includes the cost of parsing, offset resolution, and field access in the native programming language.
- **Memory Usage**: The runtime memory requirements during serialization and deserialization, including:
  - Stack usage

  - Heap usage (if any)
  - Temporary buffers or scratch memory

- **Compatibility Properties**: The support for forward and backward compatibility, including schema evolution behavior, optional fields, and handling of missing or unknown fields.

These parameters were selected based on prior empirical studies of serialization frameworks and reflect constraints commonly encountered in embedded energy systems, industrial devices, and microcontroller-based platforms.

4.2 Experimental Design

The experiments were carefully curated to isolate individual costs and minimize measurement noise. All benchmarks were conducted using equivalent logical data models and access patterns across frameworks. Where applicable, generated code was used rather than hand-written adapters, reflecting real-world usage.

To enable fine-grained analysis, the experiments were divided into the following categories:

- **Baseline Overhead**: A minimal program containing only framework utilities and generated code, used to determine default code size and static memory overhead.
- **Deserialization Only**:
  - Deserialization without field access
  - Deserialization with single-field access
  - Deserialization with full-field access
  - Deserialization with nested-field access
- **Serialization Only**:
  - Serialization without field access
  - Serialization with single-field access
  - Serialization with full-field access
  - Serialization with nested-field access
- **Combined Serialization and Deserialization**:
  - Serialization followed by deserialization without field access
  - Serialization and deserialization with single-field access
  - Serialization and deserialization with full-field access
  - Serialization and deserialization with nested-field access

This decomposition allows the separation of pure encoding/decoding costs from access-related overhead and highlights the impact of nested structures and variable-length fields.

4.3 Test Data Structures

All experiments were conducted using representative data structures derived from real-world SunSpec models. These structures include a mix of fixed-length primitive fields, variable-length fields, nested structures, and optional elements to reflect realistic energy-sector workloads.

4.4 Experimental Platforms

To evaluate behavior across both high-performance and constrained environments, all experiments were executed on the following platforms:

- **Desktop System**: Dell Inc. Precision 5530, Intel® Core™ i9-8950HK (12 threads), Ubuntu 24.04.3 LTS (64-bit)
- **Embedded System**: RP2350 microcontroller, dual Arm Cortex-M33 at 150 MHz, 520 KB RAM, 32-bit architecture

On both platforms, identical logical benchmarks were executed. Compiler optimizations were fixed across experiments, and measurements were repeated multiple times to ensure consistency. Timing measurements exclude I/O and logging overhead and focus solely on serialization-related computation.

4.5 Measurement Considerations

To reduce noise and ensure reproducibility:

- Warm-up iterations were executed prior to timing measurements.
- Heap allocations were tracked explicitly to detect dynamic memory usage.
- Stack usage was measured using static analysis and runtime watermarking on embedded targets.
- All serialized buffers were validated for correctness after each experiment.

This methodology enables a fair, repeatable, and implementation-independent comparison of serialization frameworks under conditions representative of real-world energy and embedded systems.

5 Results and Discussion

6 Conclusion