SFProto

Seghter ter Braak - 4909682
Michel Beeren - 5422833
Julia Pille - 5303117

January 30, 2026

1 Introduction

This project, referred to as SFProto (SimpleFeatures Proto), explores the use of a custom Protocol Buffers (Protobuf)-based encoding for SimpleFeatures and evaluates how it compares to widely used geospatial formats such as GeoJSON and FlatGeobuf. We focus on file size efficiency when representing two-dimensional geospatial data that includes both geometry and attributes.

Protobuf is a language-neutral, platform-neutral mechanism developed by Google for serializing structured data. It provides a compact and efficient alternative to text-based formats such as JSON or XML. In Protobuf, the structure of the data is defined once in a schema file, from which source code can be automatically generated for multiple programming languages. This generated code is used to serialize data into a binary format for storage or transmission and to deserialize it back into structured objects. Because Protobuf produces small files, enforces strict data typing, and supports fast read and write operations, it is widely used in distributed systems, cross-language APIs, and performance-sensitive applications [1].

Similar in purpose but different in design, FlatBuffers (also developed by Google) is another efficient, cross-platform binary serialization library for performance-critical applications. It enables data to be accessed directly from its binary representation without parsing or unpacking, reducing memory overhead and latency. FlatBuffers support multiple programming languages, are open source, and allow schema evolution while preserving backward and forward compatibility [2]. Building upon this foundation, the FlatGeobuf format applies FlatBuffers to the storage of geospatial data. Inspired by formats like Geobuf and Flatbash, FlatGeobuf omits support for random writes to keep the design lightweight and introduces an optional spatial indexing mechanism based on a packed Hilbert R-tree. This enables fast bounding-box spatial queries directly within the file while still allowing the format to be efficiently written as a stream or appended in cases where spatial filtering is unnecessary. By combining compactness with efficient access patterns, GeoFlatBuffers demonstrates how general binary serialization methods can be adapted for the specific demands of large geospatial datasets [3].

In contrast to these binary approaches, JSON (JavaScript Object Notation) remains one of the most common text-based data formats [4]. It is easy for machines to process, human-readable, and widely supported across programming languages, which led to the development of GeoJSON. Its geospatial extension, GeoJSON, is designed to represent geographic features and their attributes in a straightforward and interoperable way. GeoJSON combines geometry and non-spatial attributes into ”Features” and collections of features into ”FeatureCollections.” Because GeoJSON represents data in text form and often repeats structural information, it is easy to process and interpret but results in relatively large file sizes compared to binary formats [5, 6].

Underlying many of these formats is the SimpleFeatures model, formally defined by the Open Geospatial Consortium (OGC) and the International Organization for Standardization (ISO) in the SimpleFeature Access (SFA) specification. This standard defines a consistent structure for representing vector-based geographic features by combining geometry with associated attribute information [7, 8]. The SimpleFeatures model supports a fixed set of geometry types, ranging from simple primitives such as points and line strings to more complex multi-geometries and geometry collections. An overview of the geometry types is given in Table 1, while Figure 1 illustrates these geometries visually.

GeoJSON and FlatGeobuf both build upon this model, though they differ significantly in representation: GeoJSON encodes data as human-readable text, while FlatGeobuf implements a binary, columnar structure optimized for compact storage and fast spatial queries. In FlatGeobuf, the file itself acts as a FeatureCollection, beginning with a header that defines the schema and optional spatial index; each record represents a Feature, storing a binary-encoded geometry alongside its attribute values.

Table 1: SimpleFeatures model, derived from [7]

Geometry Type	**Description**
Point	Single point
LineString	Sequence of connected points forming a line
Polygon	Closed sequence of connected points forming a polygon, possibly with one or more holes
MultiPoint	Set of points
MultiLineString	Set of line strings
MultiPolygon	Set of polygons
GeometryCollection	Set of geometries of any type except GeometryCollection

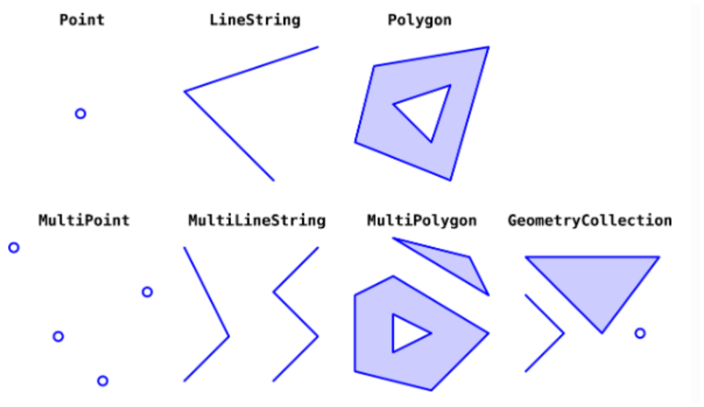

Figure 1: SimpleFeatures visualization, derived from [6]

Geospatial datasets are growing as higher spatial resolution, denser sampling, and longer time coverage are used. Because of this growth, keeping file sizes small has become very important. Common formats like GeoJSON are easy to use and read, but they produce large files because they store data in text form. Newer binary formats, such as FlatGeobuf, can make files much smaller through compact binary encoding. However, FlatGeobuf adds extra information for spatial indexing, which increases file size slightly and may not always be needed if the main goal is to simply reduce file size.

Protobuf is a general-purpose binary serialization framework designed for efficient data storage and transmission. FlatGeobuf, in contrast, is a geospatial format built on FlatBuffers that defines a complete storage layout for vector features and may include additional structures such as spatial indexing. By implementing a custom Protobuf encoding for SimpleFeatures, this study investigates how a general-purpose binary serialization framework compares to established geospatial file formats such as GeoJSON and FlatGeobuf in terms of file size efficiency.

As introduced earlier, this project investigates how a custom Protobuf-based encoding of SimpleFeatures, referred to as SFProto, compares to established formats such as GeoJSON and FlatGeobuf, with a particular focus on file size efficiency. The objective is to analyze and understand the characteristics, strengths, and limitations of the proposed encoding by encoding two-dimensional geospatial data with both geometries and attributes and comparing the results to equivalent representations in GeoJSON and FlatGeobuf.

In Chapter 2, we describe the design of SFProto, the chronological development of its versions, and the test data used in the experiments. Chapter 3 presents and evaluates the experimental results, including comparisons between different encoding variants across file size, CPU, I/O, and end-to-end performance metrics. Chapter 4 concludes the study, while Chapter 5 discusses the findings and outlines directions for future work.

2 Methodology

This chapter describes the design and evolution of all the encoding versions developed in this project. The methodology follows an iterative approach, where each version introduces specific design choices or optimizations while building on insights gained from earlier versions.

Figure 2 illustrates the chronological development of the versions, showing how the encoding strategies evolve over time. To support the interpretation of this figure, Table 2 provides an overview of the key properties of each version. In particular, the table summarizes whether a version supports attribute encoding and which schema type it is, and whether it applies delta and integer-based geometry encoding, and also here there are two options. The table therefore serves as a legend for Figure 2, allowing the reader to quickly identify which technical features are introduced, modified, or omitted in each version.

Together, the figure and table provide a clear overview of the methodological progression. The methodologies, used for these versions are explained below. First, a baseline method for encoding and decoding SimpleFeatures using Protobuf is presented. Next, the approach for encoding and decoding attributes is described, covering both static and dynamic attribute representations. Finally, the optimization steps applied to improve the storage efficiency of encoded SimpleFeatures are described.

Table 2: An overview of the versions and their properties

	Attribute encoding	Delta and integer geometry encoding
v1	✗	✗
v2	✗	✓*
v3	✓$^{\circ}$	✓*
v4	✓$^{\circ\circ}$	✗
v5	✓$^{\circ\circ}$	✓*
v6	✗	✓**
v7	✓$^{\circ\circ}$	✓**

$^{\circ}$ A static schema is used for attribute encoding.
$^{\circ\circ}$ A dynamic schema is used for attribute encoding.
* Multiple absolute points are supported. Delta encoding is used only for each LineString or polygon ring.
** One point is stored as an absolute coordinate, and the remaining points are stored as relative coordinates.

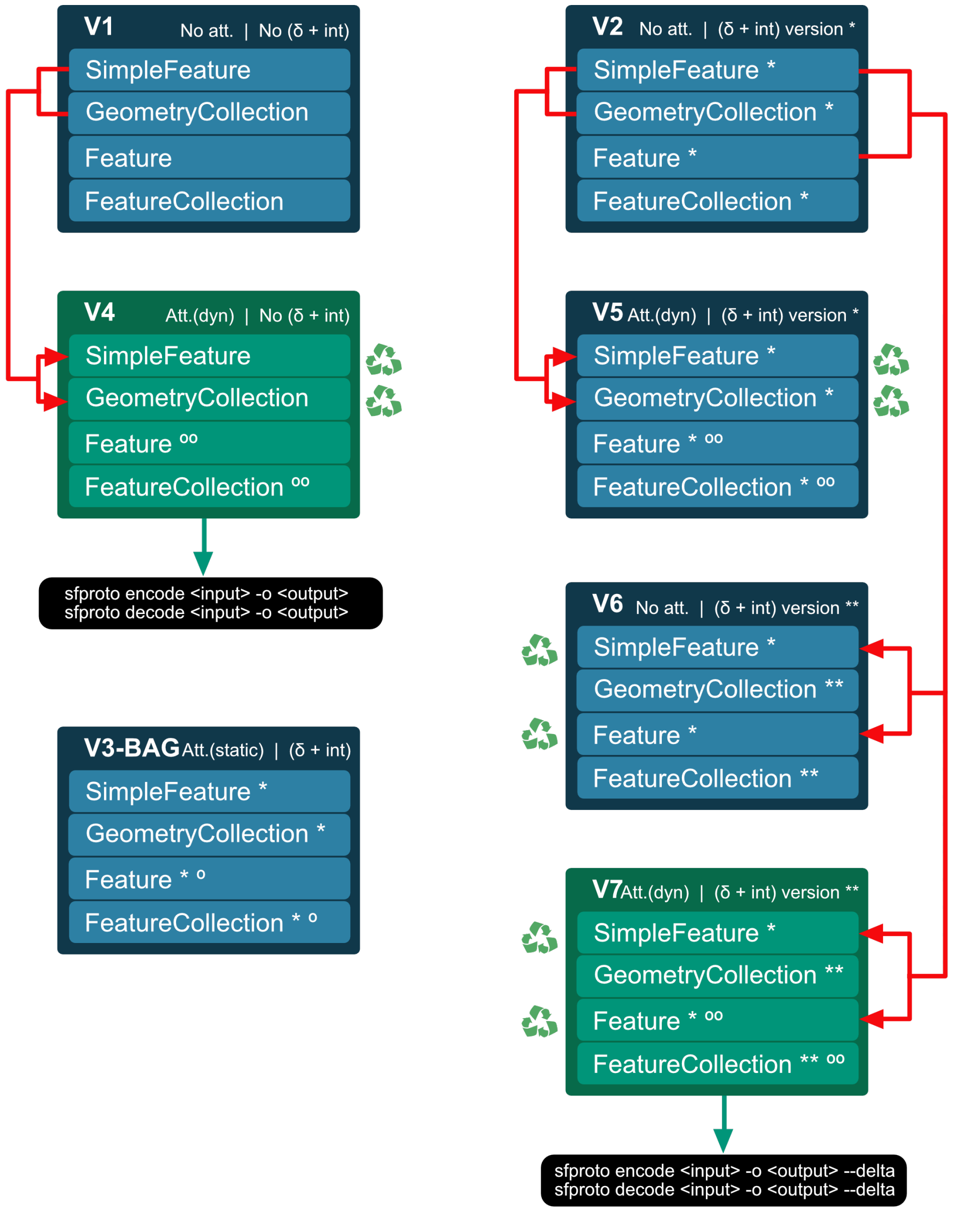

Figure 2: Chronological development of the versions, V4 and V7 are the implementations used in the command line interface.

2.1 Encoding SimpleFeatures

The encoding follows the Simple Features geometry model used by GeoJSON, in which each geometry type (e.g. Point, LineString, Polygon) has a well-defined hierarchical structure. This structure is directly mirrored in the protobuf schema, ensuring a clear and reversible mapping between GeoJSON and protobuf representations.

2.1.1 Geometry mapping

Each GeoJSON geometry type is mapped to a corresponding protobuf message, using pre-defined messages.
For `SimpleFeature` encoding without optimization/attributes, this includes:

```
message Crs { uint32 srid = 1; }
message Coordinate { double x = 1; double y = 2; }
message Point { Coordinate coord = 1; }
message MultiPoint { repeated Point points = 1; }
message LineString { repeated Point points = 1; }
message MultiLineString { repeated LineString line_strings = 1; }
message LinearRing { repeated Coordinate coords = 1; }
message Polygon { repeated LinearRing rings = 1;//[outer, hole1,...]  }
message MultiPolygon { repeated Polygon polygons = 1; }
```

These messages are then wrapped into a single geometry message, which uses a `oneof` field to indicate the active geometry type:

```
message Geometry {
  Crs crs = 1;
  oneof geom {
    Point point = 10;
    LineString line_string = 11;
    Polygon polygon = 12;
    MultiPolygon multipolygon = 13;
    MultiLineString multilinestring = 14;
    MultiPoint multipoint = 15;
  }
}
```

Example *Input GeoJSON* → Polygon with one outer ring and one inner ring (hole):

{"type": "Polygon", "coordinates": [[A_1, A_2, A_3, A_1],[B_1, B_2, B_3, B_4, B_1]]}

Each coordinate A_i and B_i represents a two-dimensional position (x, y).

Protobuf encoding → The polygon is encoded using the predefined geometry messages. First, each coordinate is stored as a `Coordinate` message, which are grouped into `LinearRing` messages. These rings are then combined into a `Polygon` message, which is wrapped inside a `Geometry` message:

```
Geometry {
  crs { srid = 4326 }
  polygon {
    rings {
      coords: [A_1, A_2, A_3, A_1]
    }
    rings {
      coords: [B_1, B_2, B_3, B_4, B_1]
    }
  }
}
```

This encoding preserves the hierarchical structure of the original GeoJSON geometry.
It allows for lossless round-trip conversion of SimpleFeatures between GeoJSON and Protocol Buffers and serves as the baseline for subsequent geometry optimizations and the inclusion of attributes.

2.2 Encoding attributes

The number and type of attributes can differ for each GeoJSON input file. The GeoJSON specification does not define strict rules for which attributes may be present, meaning that in practice the set of possible attributes is unlimited. As a result, both individual features (standalone or within a `FeatureCollection`) and the `FeatureCollection` itself may contain arbitrary attributes.
To handle this variability, two approaches for attribute encoding were implemented: a *static schema* and a *dynamic schema*.

2.2.1 Static attribute encoding

The goal of a static schema is to define all attributes in advance, before encoding the GeoJSON file. Since it is not possible to define a static schema for all possible GeoJSON inputs, this approach can only be applied to a specific dataset.
In this work, a static schema was created for the *2D BAG* dataset. A *2D BAG* `FeatureCollection` containing approximately 50,000 features was analyzed to identify all attribute keys present in the data, since the data schema was not found. These attributes were then explicitly defined in the Protobuf schema. Attributes that do not occur in the analyzed dataset will therefore not be encoded.
An advantage of a static schema is that attribute values can be optimized during encoding. Frequently occurring values can be represented more compactly. The following improvements for the static schema were made:

- **Domain-specific geometry assumption** Restricting the schema to polygon geometries removes the need for generic geometry containers and runtime type handling.
- **Fixed coordinate precision** A global, schema-defined scaling factor enables integer quantization and delta encoding without storing per-feature precision metadata.
- **Implicit structural conventions** Known properties such as polygon ring closure are omitted from storage, reducing redundancy.
- **Strongly typed attributes** Attributes are encoded as fixed fields and enums instead of dynamic key–value pairs, eliminating repeated keys and type tags.
- **Compact identifiers** Feature identifiers are stored in fixed-size binary form rather than variable-length strings.

2.2.2 Dynamic schema

The goal of a dynamic schema is to encode attributes without knowing their structure in advance. This is implemented using `struct.proto`, which is Protocol Buffers' built-in method for representing dynamic, JSON-like data.
A `Struct` is comparable to a JSON object of the form {`"field": "value"`}. Attribute names are always stored as strings, while values may be of type `"null"`, `"number"`, `"string"`, `"boolean"`, `"object"`, or `"array"`.
One limitation of this approach is that numeric values are always stored as floating-point numbers. As a result, integer values are automatically converted to floating-point values during encoding.

Example {`"bouwjaar": 1909`} is encoded as {`"bouwjaar": 1909.0`}. This conversion does not result in any loss of precision, but it slightly increases the byte size when the data is encoded and decoded.

While a `Struct` can represent almost any attribute, commonly used attributes are still defined as separate fields in the Protobuf schema (e.g. properties, name, bbox). This has several advantages:

- **Smaller file size:** common attribute names such as `"bbox"` do not need to be stored as strings for every feature.
- **Possibility for further optimization:** predefined attributes allow additional optimizations, although these were not implemented in this version.
- **Simpler and faster access:** typed fields can be accessed directly, without string look ups or runtime type checks.

It should be noted that attributes are stored according to a predefined internal structure. As a result, the order of attributes may change during serialization and subsequent de serialization.

Example *Input GeoJSON*:
`{'type': 'Feature', 'att 1': 'a', 'att 2': 'b', 'geometry': {...}}`

Decoded output:
`{'type': 'Feature', 'geometry': {...}, 'att 2': 'b', 'att 1': 'a'}`

Although the decoded output contains the same information, the order of attributes may differ from the original input.

2.3 Storing SimpleFeatures more efficiently

To encode geometries more efficiently, we store all coordinates as integer values and represent positions relative to an absolute starting point using delta encoding.

2.3.1 Floating-point to integer conversion

In GeoJSON, coordinates are typically stored as floating-point numbers. However, integer values generally require fewer bytes, which can reduce the overall size when geometries are stored in protobuf's binary format. To achieve this, floating-point coordinates are multiplied by a scaling factor and then rounded to the nearest integer. For encoding, this is defined as:

$$Coord[\text{int, int}] = round(scaler * Coord[\text{float, float}])$$

For decoding, the inverse operation is applied:

$$Coord[\text{float, float}] = \frac{Coord[\text{int, int}]}{scaler}$$

Rounding inevitably introduces a loss of precision. The magnitude of this loss strongly depends on the chosen scaling factor: higher scaling factors result in higher positional accuracy. For GeoJSON data, we assume that centimetre-level accuracy is sufficient in practice. However, the scaling factor required to achieve this accuracy depends on the SRID used by the GeoJSON.

Choosing an appropriate scaling factor
To address this, a function was implemented to automatically determine an appropriate scaling factor based on the GeoJSON's SRID. First, the SRID is extracted from the input data; if no SRID is provided, the GeoJSON default value of EPSG:4326 is assumed. Next, the SRID is classified as either geographic (latitude/longitude) or projected (linear units; metres or feet).

- For geographic SRIDs, a scaling factor of 10,000,000 is used. It should be noted that the physical distance represented by one degree varies with latitude (e.g. distances between 1 degree are much shorter near the poles than at the equator). Ideally, the scaling factor would therefore also depend on geographic location. For simplicity, however, a single global scaling factor is used, providing approximately centimetre-level accuracy across most of the Earth's surface.
- For projected SRIDs expressed in metres, a scaling factor of 100 is applied.
- For projected SRIDs expressed in feet, a scaling factor of 3048 is used.

This approach provides a consistent trade-off between storage efficiency and positional accuracy across different coordinate reference systems.

2.3.2 Delta encoding

The goal of delta encoding is to store only a single absolute coordinate per GeoJSON input, while representing all remaining coordinates as relative positions from this starting point in the encoded binary format. This approach minimizes the magnitude of stored coordinate values and improves overall storage efficiency.
In the current implementation, this strategy is fully supported for Point, LineString, GeometryCollection, and FeatureCollection inputs. For MultiPoint, MultiLineString, Polygon, MultiPolygon, and Feature GeoJSON inputs, multiple absolute coordinates may still be present during encoding.
In an earlier implementation (v2, see Figure 2), delta encoding was applied at the level of individual SimpleFeatures. More specifically, delta encoding was only used for LineStrings and Linear Rings

(from which Polygons are constructed). This already resulted in encoding with one absolute coordinate for Point, and LineString GeoJSON inputs. However, when extending the approach to use a single absolute coordinate per GeoJSON input, only a modest percentage wise reduction in byte size was observed for FeatureCollections (approximately 4% for the 2D BAG dataset).
As a result, the focus shifted to improving delta encoding for GeoJSON inputs that typically contain a large number of coordinates, namely GeometryCollection and FeatureCollection.

Flattened coordinate representation
The improved approach starts by extracting the first coordinate of the GeoJSON input and storing it as an absolute reference point. All remaining coordinates from all contained geometries are then flattened into a single array of relative coordinate differences:

$$[\delta x_0, \delta y_0, \delta x_1, \delta y_1, ..., \delta x_n, \delta y_n]$$

At the same time, structural metadata is recorded to preserve the original geometry layout. For each geometry, the number of parts and their respective sizes are stored:

- number of points in a MultiPoint
- number (and length) of LineStrings in a MultiLineString
- number (and length) of rings in a Polygon
- for MultiPolygons, two counters are required: (1) number (and length) of rings per Polygon, and (2) total number of Polygons.

Example
Input GeoJSON $\rightarrow$ MultiPolygon consisting of:

- Polygon 1: rings (A_1, A_2, A_3, A_1) and (B_1, B_2, B_3, B_4, B_1)
- Polygon 2: ring (C_1, C_2, C_3, C_1)

Encoded format $\rightarrow$ Flattened coordinates:
(A_1, A_2, A_3, A_1, B_1, B_2, B_3, B_4, B_1, C_1, C_2, C_3, C_1)

Metadata:

- ring_sizes: [4, 5, 4]
- polygon_ring_counts: [2, 1]

This representation indicates that:

- the first 4 coordinates form a ring
- the next 5 coordinates form a ring
- the next 4 coordinates form a ring
- the first two rings belong to the first polygon
- the final ring belongs to the second polygon

A limitation of this approach is that random access to individual features is not possible. For example, decoding feature 20 requires decoding all preceding features, as the delta-encoded coordinate stream depends on the accumulated relative positions of earlier geometries.

2.4 Test Data

The evaluation was conducted on a combination of real-world and synthetic GeoJSON datasets, chosen to cover a wide range of geometric and attribute characteristics. All datasets were stored and processed in GeoJSON format to ensure a consistent input representation across experiments.

2.4.1 BAG-2D

Dutch BAG pand geometries were obtained via the PDOK WFS service. This dataset represents a highly regular, polygon-heavy geometry with a well-defined and stable attribute schema.

2.4.2 Overture Maps (buildings)

Building geometries were extracted from the Overture Maps release using DuckDB with the spatial and HTTPFS extensions. Parquet tiles were queried directly from the public S3 bucket and converted to GeoJSON. A fixed subset of 100,000 building geometries was selected to provide a globally sourced, geometry-focused dataset with minimal attribute content. This dataset was only used for testing, but since it is comparable to the others in characteristics, no specific tests were conducted with it.

2.4.3 OpenStreetMap (OSM)

OSM data covering the full country of Luxembourg was used as the basis for controlled synthetic benchmarks. This dataset was chosen for its heterogeneity in both geometry types and attributes. To enable fair comparison with FlatGeobuf, all GeoJSON datasets were converted to FlatGeobuf using ogr2ogr, explicitly disabling spatial indexing.

From the OSM source, a systematic dataset generation procedure was applied to construct benchmark inputs along three controlled dimensions:

- **Feature count:** 10, 100, 1,000, 10,000, and 100,000 features
- **Geometry regime:**
  - *Mixed*: equal proportions of points (33%), lines (33%), and polygons (33%)
  - *Geometry-heavy*: dominated by polygons (70%)
  - *Attribute-heavy*: dominated by points (80%)
- **Attribute profile:**
  - *None*: no attributes
  - *Few*: minimal identifiers
  - *Medium*: identifiers and names
  - *Many*: full attribute set

Geometry proportions were enforced deterministically, and attributes were selected from a fixed schema to ensure reproducibility. For each combination of size, geometry regime, and attribute profile, a separate GeoJSON FeatureCollection was generated.

This methodology allows isolating the effects of geometry complexity, attribute payload, and dataset scale on serialization size and performance, while keeping the underlying data source and generation process consistent.

3 Results

3.1 Relative Representation Size of GeoJSON, SFProto and FlatGeobuf

To assess storage efficiency, a series of synthetic GeoJSON datasets were benchmarked against three alternative representations: sfproto v4, sfproto v7, and FlatGeobuf without spatial index. Dataset sizes range from 10 to 100,000 features and vary along two dimensions: geometry complexity (geometry-heavy vs. attribute-heavy) and attribute payload (none, few, medium, many) (See Chapter 2.4 for the specifics). File sizes are reported relative to compact GeoJSON (size ratio = encoded size / GeoJSON size), such that values below 1 indicate a smaller representation.

3.1.1 Relative Size vs. Number of Features

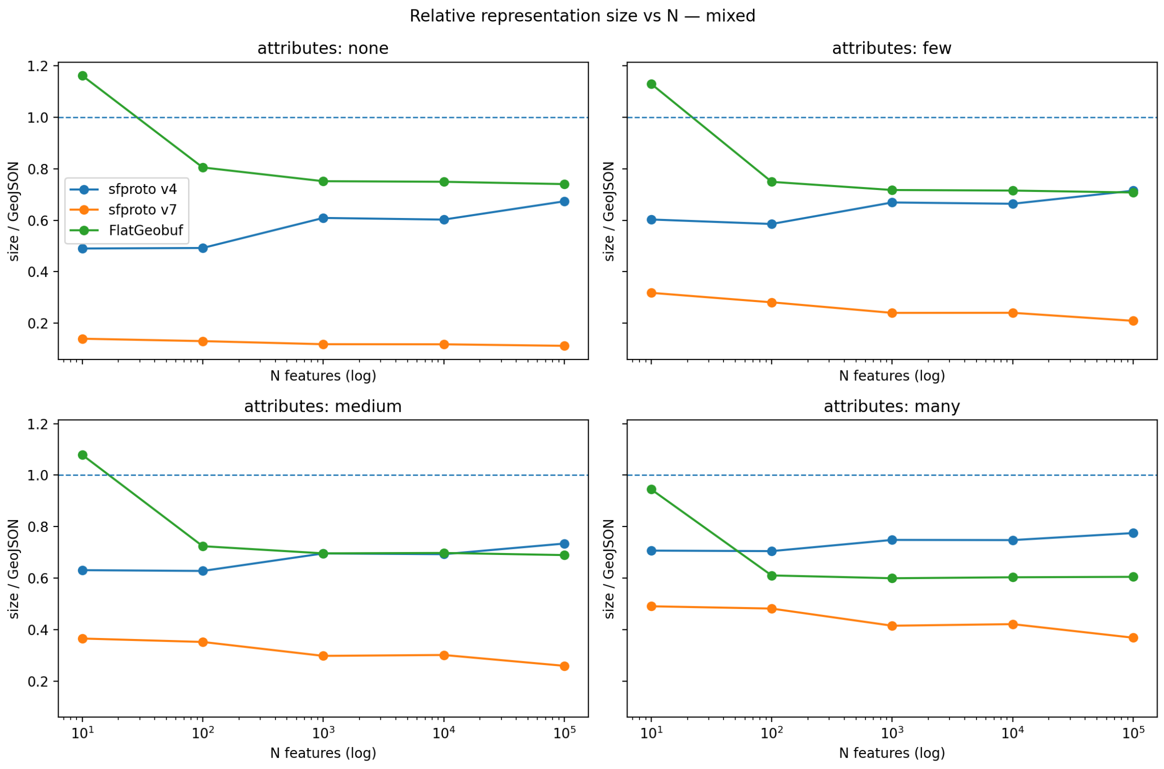

Figure 3: Relative size vs. number of features

Across all attribute profiles, FlatGeobuf shows a decrease in relative size as the number of features increases, with the strongest reduction between 10 and 1,000 features (Figure 3). Beyond this range, its relative size stabilizes as fixed metadata is spread over more features.

The relative size of sfproto v4 generally increases with the number of features. Because sfproto v4 stores absolute geometry values per feature, its size grows steadily, while GeoJSON benefits from repeated structure and similar coordinate values that slightly reduce its average size per feature as datasets grow.

sfproto v7 remains largely stable across feature counts, as storing coordinate differences instead of absolute values reduces the per-feature geometry cost and counterbalances the relative improvement observed for GeoJSON.

As attribute payload increases, FlatGeobuf performs comparatively better than the sfproto variants. Its column-oriented storage stores attribute definitions once and shares them across all features, making it increasingly efficient for attribute-heavy datasets even without the use of delta encoding. Consequently, the relative gap between FlatGeobuf and the sfproto encodings narrows as attribute payload grows.

3.1.2 Effect of Attribute Payload on Representation Size

For N = 100,000 features, increasing the attribute payload affects the formats in different ways. sfproto v7 consistently produces the smallest representation across all dataset types and attribute profiles. Its relative size increases with larger attribute payloads but remains well below sfproto v4 and FlatGeobuf. This efficiency comes from delta encoding and coordinate quantization, which introduce a controlled loss of geometric precision. But if you interpret these lines as a trend, then FlatGeobuf is like to dominate when the data is really attribute heavy.

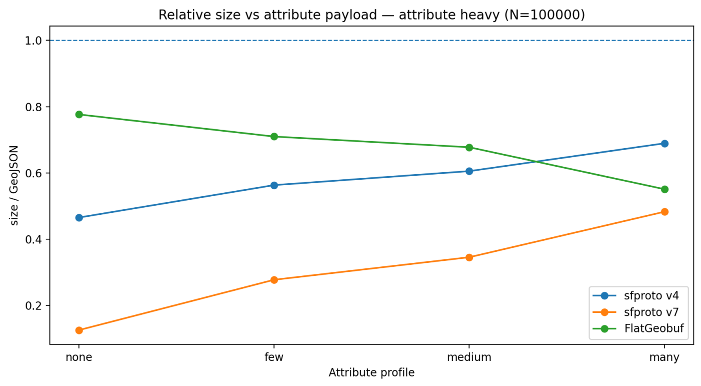

Figure 4: Relative size vs. attribute payload, on an attribute heavy dataset

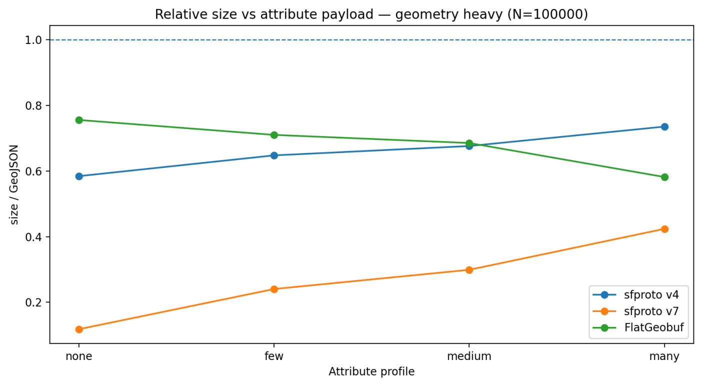

Figure 5: Relative size vs. attribute payload, on a geometry heavy dataset

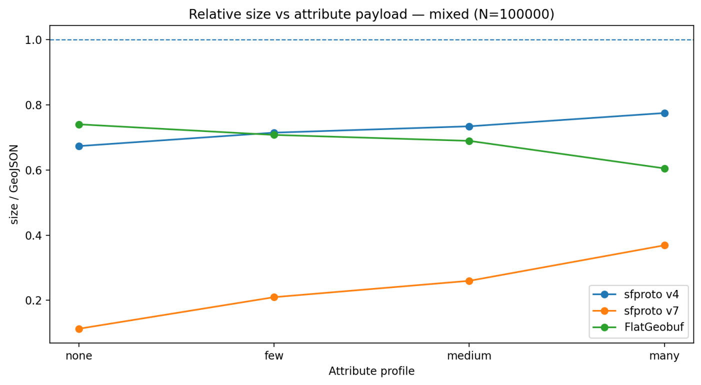

Figure 6: Relative size vs. attribute payload, on a geometry mixed dataset

sfproto v4 shows a steadily increasing relative size as attribute payload grows, most notably in geometry-heavy and mixed datasets, where it approaches or exceeds the size of FlatGeobuf for larger attribute payloads.

FlatGeobuf exhibits the opposite trend: its relative size decreases as attribute payload increases. While it has higher overhead for datasets with few or no attributes, it becomes increasingly efficient for attribute-heavy datasets and outperforms sfproto v4 for medium and many attributes, despite not using delta encoding.

3.2 2D-BAG: Static vs. Dynamic Schema

The following benchmarks were performed on a BAG pand FeatureCollection with 10,000 features. Three binary encodings were compared:

- BAG v3: static, domain-specific schema
- v4: dynamic schema
- v7: dynamic schema with delta encoding and quantized coordinates

All results are reported as means over 200 repeated runs.

3.2.1 Serialized size

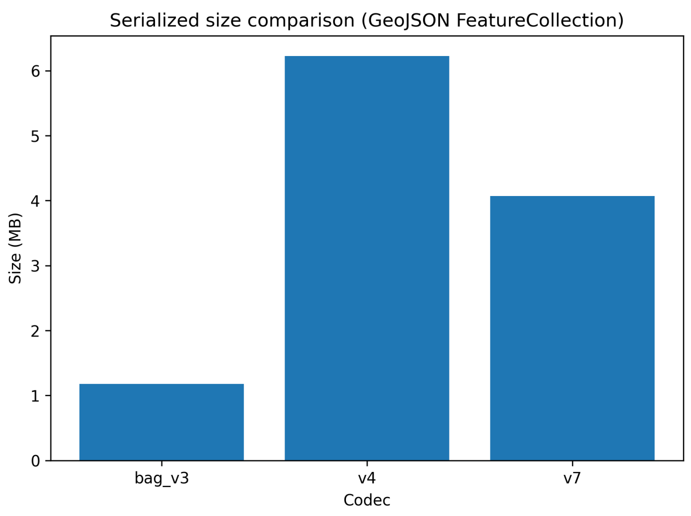

Figure 7: Serialized size comparison of binary encoding versions

The serialized size comparison (Figure 7) shows a clear hierarchy. The static BAG v3 schema produces the smallest representation by a wide margin, followed by sfproto v7, while sfproto v4 results in the largest binary size.

The reduction from v4 to v7 can be directly attributed to the use of integer quantization and delta encoding in v7. By storing coordinates as scaled integers and encoding only differences between consecutive points, v7 exploits the strong geometric locality of the BAG dataset, resulting in substantially lower per-coordinate storage. This comes at the cost of a controlled loss in geometric precision.

Although both v4 and v7 support dynamic attributes via `struct.proto`, the results show that schema flexibility alone does not ensure compactness. In v4, geometries are stored as absolute double-precision values per feature, leading to higher overhead. In contrast, v7 combines dynamic attributes with structural compression, significantly reducing the serialized size.

The static BAG v3 schema still outperforms both dynamic approaches, indicating that domain-specific schema design enables additional optimizations that are not achievable with fully generic GeoJSON-style encodings.

3.2.2 CPU performance: encode vs. decode

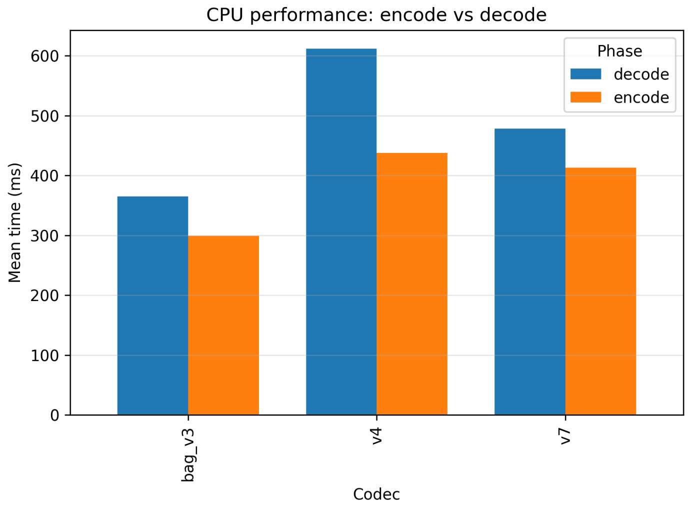

Figure 8: CPU performance on encode vs. decode between the three versions

The CPU benchmark (Figure 8) shows that decoding is consistently more expensive than encoding across all formats. sfproto v7 improves both encode and decode performance relative to sfproto v4, but remains slower than the static BAG v3 schema.

sfproto v4 exhibits the highest decode cost due to its nested geometry structure and use of absolute double-precision coordinates, which increase parsing work and memory traffic. In contrast, sfproto v7 benefits from a flatter geometry layout and a smaller serialized representation, reducing the amount of data that must be parsed and reconstructed.

Delta decoding in v7 introduces additional arithmetic to recover absolute coordinates, but this cost is outweighed by reduced parsing and object creation overhead. A similar effect is observed during encoding: although delta encoding requires extra computations, the reduced output size leads to lower serialization and memory-write costs.

BAG v3 remains the fastest in both phases, reflecting the efficiency of its static, domain-specific schema without dynamic attribute handling.

3.2.3 Raw disk IO throughput

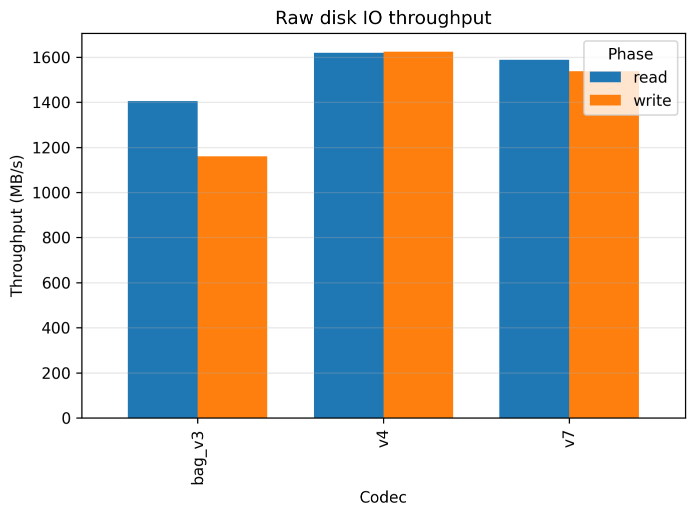

Figure 9: Raw disk IO throughput

The raw IO throughput results (Figure 9) primarily reflect sequential disk access rather than encoding efficiency. sfproto v4 shows the highest throughput, followed closely by sfproto v7, while BAG v3 performs slightly lower.

The higher throughput of v4, despite its larger size, results from efficient sequential reads and writes handled by the operating system. Since no parsing or decoding occurs in this benchmark, format-level optimizations such as delta encoding do not influence the results.

As a result, raw IO throughput alone is not representative of overall performance; meaningful differences only emerge when serialization and deserialization are included.

3.2.4 End-to-end performance (serialize + IO)

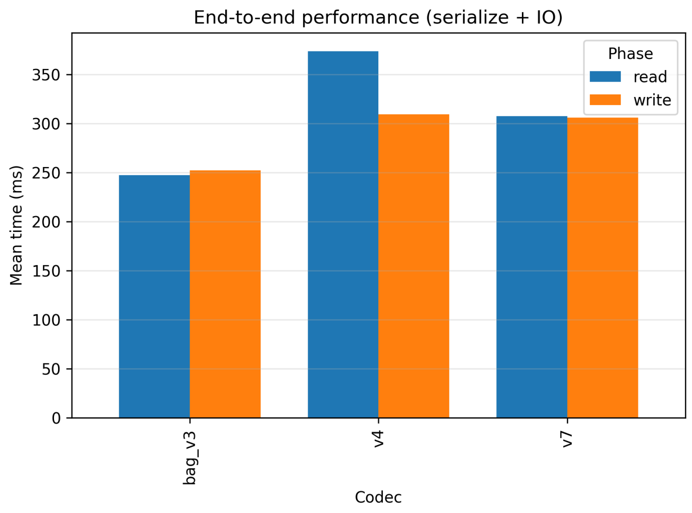

Figure 10: End to end times performance

The end-to-end benchmark (Figure 10) provides the most practically relevant comparison, as it combines serialization, disk IO, and deserialization.

sfproto v7 consistently outperforms sfproto v4 for both read and write operations. This confirms that the reduction in serialized size and the flatter, delta-encoded geometry layout lead to lower overall system cost, despite the additional computations required to reconstruct coordinates.

BAG v3 achieves the best end-to-end performance, most notably for read operations. Its compact, static schema minimizes both IO volume and decoding overhead, allowing it to outperform the dynamic approaches.

Overall, the results show that delta encoding significantly improves end-to-end performance within a dynamic schema, but does not fully match the efficiency of a specialized, domain-specific encoding when such a schema is available.

4 Conclusion

This project set out to design and evaluate a custom binary encoding for SimpleFeatures using Protocol Buffers, with the goal of understanding the trade-offs of different serialization strategies and comparing them to established geospatial formats such as GeoJSON and FlatGeobuf. By iteratively developing multiple SFProto variants and benchmarking them on real-world and synthetic datasets, the study provides practical insight into how schema design, geometry encoding, and attribute handling affect storage efficiency and performance.

The results show that a straightforward Protobuf mapping of GeoJSON structures (sfproto v4) already achieves substantial size reductions compared to GeoJSON, but suffers from increased overhead as dataset size and attribute payload grow. Introducing integer quantization and delta encoding (sfproto v7) significantly improves both serialized size and end-to-end performance. In particular, the flattened, delta-encoded geometry representation reduces per-coordinate storage and parsing costs, resulting in consistent gains across file size, CPU performance, and combined serialization–IO benchmarks. These improvements come at the cost of a controlled loss in geometric precision, which is acceptable for many practical applications but must be considered explicitly.

Comparisons with FlatGeobuf highlight an important distinction between general-purpose and domain-specific encodings. FlatGeobuf performs increasingly well for attribute-heavy datasets due to its column-oriented layout, even without delta encoding, but carries fixed metadata overhead and design choices oriented toward spatial querying rather than minimal size. The static BAG v3 schema consistently outperforms all dynamic approaches, demonstrating that domain-specific schema design enables optimizations that cannot be achieved with fully generic SimpleFeatures encodings.

Overall, this work confirms that Protocol Buffers can be used effectively to encode SimpleFeatures and that structural compression techniques such as delta encoding and coordinate quantization are crucial for achieving competitive performance. At the same time, the results underline that no single format is optimal for all use cases: dynamic schemas offer flexibility, while static, domain-specific schemas deliver superior efficiency when the data model is known in advance. The SFProto implementation and benchmarks provide a reproducible basis for further exploration of hybrid approaches, alternative encoding strategies, and extensions toward streaming or query-oriented geospatial data formats.

5 Discussion

This study shows that serialization performance for geospatial data is largely determined by schema design and geometry encoding strategy rather than the choice of serialization framework alone. Moving from sfproto v4 to v7 demonstrates that integer quantization, delta encoding, and a flatter geometry layout substantially reduce serialized size and improve CPU and end-to-end performance, at the cost of controlled precision loss.

The static BAG v3 schema consistently outperforms the dynamic approaches, highlighting the advantage of domain-specific schemas. Fixed geometry types and attribute definitions enable optimizations that are not achievable in generic SimpleFeatures encodings, regardless of the underlying framework.

Fair comparison with GeoJSON is inherently difficult. GeoJSON is designed for interoperability and human readability, not storage efficiency. Its textual structure benefits indirectly from repeated strings and numeric patterns, especially when external compression is applied, while its parsing cost depends heavily on the JSON tooling used. As a result, GeoJSON's performance characteristics are influenced more by surrounding infrastructure than by the format itself.

Comparisons with FlatGeobuf are similarly challenging. FlatGeobuf is optimized for spatial access and streaming rather than minimal size, and includes mandatory metadata that introduces fixed overhead, particularly for small datasets. Its column-oriented attribute layout favors attribute-heavy data, but this advantage is not fully reflected in sequential serialization benchmarks, especially with spatial indexing disabled.

Overall, the results underline that no single format is optimal for all scenarios. Dynamic binary encodings such as sfproto v7 are well suited for compact transport and storage of SimpleFeatures when precision requirements are known, while static schemas and query-oriented formats remain preferable when domain constraints or spatial access patterns dominate.

As a direction for future work, the approach could be extended to encode and decode three-dimensional datasets such as the 3D BAG. Supporting volumetric geometries would make it possible to study how the observed trade-offs apply to more complex geospatial data. In addition, incorporating a spatial indexing mechanism would be an interesting extension, enabling efficient spatial queries and allowing a more direct comparison with formats such as FlatGeobuf that integrate indexing at the storage level. Finally, combining optimized static attribute encoding for frequently occurring attributes with dynamic attribute encoding can further reduce byte size and improve read and write performance, while still encoding all attributes.

References

[1] Protocol Buffers — protobuf.dev. https://protobuf.dev/. [Accessed 29-01-2026].

[2] FlatBuffers Docs — flatbuffers.dev. https://flatbuffers.dev/. [Accessed 29-01-2026].

[3] FlatGeobuf — flatgeobuf.org. https://flatgeobuf.org/. [Accessed 29-01-2026].

[4] JSON — json.org. https://www.json.org/json-en.html. [Accessed 29-01-2026].

[5] GeoJSON — geojson.org. https://geojson.org/. [Accessed 29-01-2026].

[6] Ben-Gurion University of the Negev. GeoJSON. https://geobgu.xyz/web-mapping/geojson-1.html. [Accessed 29-01-2026].

[7] OGC API Features Standard — REST API for Geospatial Features — ogc.org. https://www.ogc.org/standards/ogcapi-features/. [Accessed 29-01-2026].

[8] OGC API - Features - Part 1: Core — docs.ogc.org. https://docs.ogc.org/is/17-069r3/17-069r3.html. [Accessed 29-01-2026].