# NakedBytes (Offset Buffer): A Unified, Compact, and Self-Referencing Offset-Oriented Serialization Framework for Embedded and High-Performance Systems

Busoye Tolulope Matthew

December 18, 2025

**Abstract**

NakedBytes is an offset-based serialization framework designed for embedded and high-performance systems that demands minimal overhead and maximal speed. It leverages zero-copy, in-place data access so that serialized objects can be used directly in memory without parsing or relocation. In practice, this means values are accessed by simple pointer arithmetic rather than deserialization. This approach follows the trend of modern zero-copy formats (e.g. FlatBuffers, Cap'n Proto) which "allow direct data access from raw binary without any deserialization"[1], yielding near-zero latency and no extra copies[2][3]. Unlike some existing systems, NakedBytes is highly compact: it eliminates table vtables and excessive padding, so that very little space is wasted[4][3]. At the same time, it supports schema evolution via built-in forward/backward compatibility checks. The format is language-agnostic and the binary representation is canonical (IEEE-754 floats, two's-complement integers, little-endian as on common CPUs[5]). The result is a simple, fast serialization that can run on microcontrollers (with a preallocated arena and no dynamic memory) as well as on servers. This paper presents the motivation, design goals, model, schema language, and technical specification of NakedBytes, and includes an example illustrating its usage.

## 1 Introduction

### 1.1 Background

Data serialization is a core concern in software systems: converting in-memory objects to a byte stream for storage or transmission, and reconstructing them back. Traditional text formats like JSON or XML are flexible but require expensive parsing and are bulky. Binary formats such as Google Protocol Buffers and Apache Thrift improve size and speed, but still require extra processing (parsing or copying) on deserialization. In contrast, recent zero-copy serialization formats enable accessing structured data directly in place, without a decode phase. Formats like FlatBuffers and Cap'n Proto expose data via pointers/offsets, so readers can traverse the in-memory binary with simple pointer arithmetic[1]. This is ideal for performance-critical and embedded use cases: for example, tracking flight telemetry or running on memory-constrained IoT devices demands "no memory copies", "near-zero latency", and minimal RAM usage[2][6]. TensorFlow Lite Micro, for instance, embeds machine-learning model data in Flash as a FlatBuffers buffer, which can be accessed directly without parsing[3].

## 1.2 Limitations of Existing Serialization Systems

Despite their advantages, existing serialization systems have trade-offs. Textual JSON/XML are human-readable but inefficient in speed and space. Protobuf and similar binary formats use variable-length encodings, requiring a parsing pass and dynamic allocation, which adds latency and memory pressure. FlatBuffers and Cap'n Proto avoid parsing, but they introduce their own overhead. For example, FlatBuffers tables use vtables (virtual tables of field offsets) which incur a fixed overhead (typically 24 bytes per table) in the binary[4]. In one observation, a FlatBuffers-based model wasted on the order of 40% of its bytes in vtables and padding. Cap'n Proto sacrifices features (no optional fields, complicated encoding, no storage of data in canonical form) to optimize speed[7]. Moreover, many systems do not naturally support both forward and backward compatibility; evolving schemas often require careful rules or reserialization. Crucially for embedded systems, many serializers assume dynamic memory allocation or only work on powerful CPUs, making them unsuitable for bare-metal or resource-limited environments. These gaps motivated the development of NakedBytes, which aims to combine the speed and simplicity of zero-copy formats with maximal compactness and full compatibility support. talk about sbe

## 1.3 Motivation

The design of NakedBytes was driven by practical needs in embedded and high-performance contexts. For example, in power systems modeling (SunSpec), using FlatBuffers revealed that a large fraction of the buffer was overhead (vtables, padding), which is unacceptable on a microcontroller. Similarly, TensorFlow Lite Micro uses FlatBuffers to represent neural network parameters, but the goal is to reduce any unnecessary footprint and simplify access. NakedBytes aims to serve as a "native database" format on microcontrollers: the serialized data is in canonical form, ready for immediate use without interpretation. Writes must be fast (just memory copies), and reads must be only a few pointer operations, so that data can be stored and accessed at high rates. The design mandates simplicity (familiar C/C++-like types), flexibility (support for complex object graphs and schema changes), and efficiency (fast access, small size, cache-friendly layout). Achieving forward and backward compatibility is also a key objective, so that as schemas evolve fields can be added/removed with minimal impact. Overall, NakedBytes is intended to replace heavy-weight or less-compact serializers in scenarios ranging from tiny IoT sensors up to servers and databases, while delivering "fast as possible" read/write and "compact as possible" storage.

## 1.4 Objectives

Based on the above, the core objectives of NakedBytes are:

1. Maximize access speed – reading a fixed-length field should require no more than a direct memory load (no decoding or multiple operations).

2. Minimize size overhead – the serialized binary should include only the necessary data, not extraneous tables or padding;

3. Support canonical in-memory representation – data is stored little-endian in IEEE754/two's-complement form to match common CPU formats[5];

4. Be simple and language-neutral – schemas look like plain C/C++ types (ints, structs, arrays, etc.), and code can be generated in multiple languages;
5. Allow zero-copy usage – deserialization is just pointer arithmetic, with no extra allocations or copies[1][3];
6. Use an arena allocator – all serialization is done in a single preallocated buffer, so no dynamic heap is needed;
7. In place serialization – Inplace serialisation allowing no need for relation of parsed data but rather data are serialed in the arane andrefence offset are made to it.
8. Enable schema evolution – the format provides built-in mechanisms so that new fields can be added (forward compatibility) and old data can still be read (backward compatibility).

The following sections detail how these principles are realized in the NakedBytes design.

## 2 Design Goals and Principles

- Compactness: The format minimizes wasted space. All fixed-size fields (integers, floats, booleans, enums, fixed arrays, and structs) are stored inline without any tagging or additional header. Variable-sized data (strings, blobs, vectors, optional objects) are stored out-of-line, referenced only by a compact offset pointer in the parent. Unlike some table-based formats, NakedBytes does not use separate vtables or bitmaps for fields, which removes their fixed overhead. For example, FlatBuffers incurs about 24 bytes of overhead per table[4]. By contrast, NakedBytes omits such tables entirely and only stores each object's raw data plus a small offset for variable parts, yielding "very little wasted space"[3]. In practice, new fields and padding are only introduced if needed for alignment or explicit optionality, and schema generators warn if field ordering causes excessive padding.
- Offset-Oriented Structure: All references to variable-length data use relative offsets. In the serialized buffer, fixed-size fields remain at constant positions, while any variable-sized field is represented by an offset (of configurable size, e.g. 8 bits up to 64 bits) that tells how far away the actual data is. An offset is a signed integer added to the current position to get the target address. For example, a string member appears as a 32-bit offset in the struct; the value of that offset is added to the string's position field in the buffer to locate the string's byte data. By using signed offsets, the data can be placed either ahead or behind, which allows flexibility during serialization (though typically data is appended forward). This offset mechanism is similar to FlatBuffers' use of soffset_t, which is also a relative pointer to a vtable[8]. In NakedBytes, we extend this idea: all variable-length fields (including vectors, blobs, nested objects, and unions) are accessed via offsets. This enables in-place zero-copy access, since computing an offset-based pointer is just a constant-time addition. It also makes the data layout cache-friendly: fixed-size portions remain tightly packed, and large blobs are stored separately at the end, reducing cache misses for small reads.
- Language Independence: The schema for NakedBytes is defined in a language-neutral way (we use JSON for the schema definition), and code is generated for the target language

(C/C++, Rust, etc.). The primitive types align with standard C types (uint8_t, int32_t, float, double, bool, etc.) and endianness is fixed to little-endian to match "all commonly used CPUs"[5]. Enumerations are just integer types under the hood. Because data is in a canonical format, any language that knows how to read these types from memory can use the buffer without a specific runtime library. In effect, NakedBytes treats the serialized buffer as a "C struct in memory"[3], with a known schema. This mirrors how FlatBuffers uses code generation from an .fbs schema to create C/C++ classes[9], except our schema is JSON for the first draft.

- Canonical Binary Representation: All primitive values are stored in binary form matching CPU conventions: little-endian byte order, IEEE-754 for floats, and two's-complement for signed integers[5]. There is no additional compression or custom encoding. The aim is that the buffer is the in-memory representation. For example, a 32-bit integer field in the struct is stored as 4 little-endian bytes at its offset. This has two benefits: it eliminates conversion cost on access, and it maximizes performance on modern hardware (which is almost always little-endian[5]). Structs impose natural alignment (each field is aligned to its size) so that CPU loads are efficient; padding is added only as needed to satisfy these alignments.

- In-Place Read/Write (Zero Copy): Reading any value is done by a simple pointer or offset calculation; no per-field parsing or dynamic memory allocation is required. For fixed-length fields (primitives, inline arrays, and structs), access is literally a direct memory reference. For variable-length fields (strings, vectors, unions, nested objects, blobs), access is computed by adding the stored offset to the base address – akin to pointer dereferencing. Thus, reading a string involves reading its stored length (an integer) and then accessing the bytes that follow (terminated by null). Because of this, users can write highly efficient code: e.g. int32_t x = *(int32_t*)(&data[MyStruct_X_OFFSET]); or an automatically generated equivalent. This zero-copy model follows established principles: "formats like FlatBuffers and Cap'n Proto allow direct data access from raw binary without any deserialization"[1], which is ideal for high-throughput or embedded scenarios. On writing/serialization, NakedBytes simply writes each field in place (or defers variable parts) – again, just memory operations. The write performance matches other similar systems, since it's essentially a memcpy into the buffer.

- Arena-Based Serialization (No Allocation Requirement): NakedBytes uses a single contiguous buffer (an arena) to build the serialized data. The caller is responsible for providing a buffer large enough for the final size (determined by offset_size limits). During serialization, all objects and values are placed into this buffer without any dynamic memory allocations or secondary buffers. This is critical for systems that cannot easily allocate heap memory (e.g. bare-metal MCUs). Because data is placed in one arena, pointer offsets remain valid within it. This design is in the spirit of many embedded serializers and even some database engines, which build in a preallocated memory arena. The result is that all memory usage during serialization is deterministic and limited to the size of the buffer. Medium workloads like repeatedly serializing object graphs can reuse the same buffer or a stack-allocated arena to avoid any new allocations, aligning with the goal that "zero-copy formats… avoid extra RAM allocation"[6].

- Forward/Backward Compatibility: NakedBytes is built to support schema evolution. For forward compatibility (newer readers reading older data), the format allows new fields to be added safely: older code will simply never see extra data, and newer code will see default values for missing fields. We follow the principle that new fields should be appended so that older readers ignore them – a strategy used by FlatBuffers[10]. For backward compatibility (new data read by old code), each class (the top-level object) records its own length of fixed (inline) data. When accessing a field, code checks whether the requested field offset is within the stored length; if it exceeds the root size, the field is treated as absent (null) even if defined in the new schema. This is an optional check used only when necessary to allow old readers to safely skip new fields. Conceptually this resembles FlatBuffers' strategy of checking vtable ranges to see if a field is present[11]. In NakedBytes, only the "class" type uses this root-length check; nullable structs simply use a 0 offset to indicate null by default. Overall, these compatibility features mean users can evolve their data types (add or deprecate fields) with fewer breakages.

## 3 Serialization Model Overview

### 3.1 Data Model Concepts

NakedBytes categorizes types into fixed-length and variable-length, based on how their data is stored. Fixed-length types have a known constant size in bytes, and their data is stored in-place within the parent object's memory. These include: - Scalar primitives: All integer types (signed/unsigned 8/16/32/64-bit), floating-point types (float32, float64), and boolean. These occupy 1, 2, 4, or 8 bytes, aligned to their size. - Enums: Defined as an integer of fixed size (e.g. a named enum with underlying int32_t). It's stored just like its base integer type. - Arrays: Static arrays of a primitive or struct type with a fixed element count. Since the count is fixed by the schema, the array occupies a contiguous block of that many elements inline (no length prefix). - Structs: Composite types (akin to C struct) whose every field is of fixed size. A struct's binary layout is simply the concatenation of its fields (with padding for alignment). Like a C struct, a NakedBytes struct is always stored inline wherever it appears. Structs do not support optional (absent) fields; every field defined in the struct must be present. Structs can contain nested structs or arrays. The advantage is that structs impose a known, compact layout (fields are tightly packed with only alignment padding). As the FlatBuffers documentation notes, "Structs define a consistent memory layout... [and] are always stored inline in their parent (a struct, table, or vector) for maximum compactness"[12]. Variable-length types are those where the size of the data is not fixed by the schema, so the data cannot be stored fully inline. Instead, the parent object stores a fixed-size offset to the actual data payload. The variable types include: - Nullable Struct: A struct that is allowed to be null (absent). It is treated like a pointer: the parent stores an offset to the struct data if present, or 0 to indicate null. The struct data itself is like a normal struct placed elsewhere in the buffer. - Class (Table): A more flexible object (like a FlatBuffers table) that may have optional fields. In NakedBytes, a "class" is essentially a nullable struct plus versioning information. It is serialized by first writing a length (of its fixed part), then writing its fields (with offsets for any nested variable parts). Classes support forward/backward compatibility and may omit fields. - Unions: A tagged union of one of several types. Represented by two adjacent values: a small integer tag (an enum of offset_type width) indicating which type is present (or NONE=0 if null), followed by an offset to the actual payload of that type. (The FlatBuffers union is implemented similarly: an enum plus an offset)[13]. - Vector: A sequence (array) of elements of the same type, but with variable length. Vectors are stored as a length prefix (an unsigned count of elements, in the same width as an offset) followed by the elements' data. The parent stores an offset to this vector blob. Elements within the vector may themselves be fixed or offset types; if they are offset types (e.g. vector of strings), then each element in the blob is an offset to the string data. In NakedBytes, vectors are never inline in the parent object – only their offset is stored in the parent. - String: A variable-length UTF-8 string. Encoded as a (non-null) length prefix plus that many bytes (followed by a null terminator if desired). The parent stores an offset to the first byte of the string data. In practice, we include both length and a null byte. Conceptually this is a special case of a vector of char. (FlatBuffers treats strings as length-prefixed, null-terminated byte vectors[14].) - Blob: A raw binary blob (byte array) of arbitrary length. Represented like a string but without null termination – i.e. a length prefix followed by raw bytes. Again, only an offset to the blob is stored in the parent. In summary, fixed-length types (primitives, enum, struct, array) are stored directly in place and accessed by direct load/store. Variable-length types (nullable struct, class, union, vector, string, blob) are accessed via a relative offset pointer. This dichotomy ensures that the parent object remains compact (only fixed parts inline, all big payloads moved to the end of the buffer).

## 3.2 Relative Offset Mechanism

At the heart of NakedBytes is the use of signed relative offsets for all variable-sized fields. Each offset has a configured bit width (8, 16, 32, or 64 bits) and is stored in little-endian format. In the serialized buffer, an offset value at byte position p means that the actual data for that field starts at address p + offset. By making offsets signed, data can be placed either before or after the offset field. In practice, NakedBytes typically serializes forward (placing offsets then data), but the signedness allows more flexibility if needed. This design mirrors FlatBuffers' use of soffset_t (a signed offset) to refer to tables and vtables[8]. The choice of offset size is a trade-off between range and space. Smaller offsets (e.g. 8-bit) yield a tighter binary but limit buffer size (max offset of 127 forward). Larger offsets (64-bit) support very large data at the cost of extra bytes per pointer. NakedBytes allows the schema to specify the offset size, adapting to the application (small microcontroller vs large server). This is similar in spirit to FlatBuffers, which fixes uoffset_t at 32 bits by default for portability[15], but here we generalize to 8–64 bits as needed. It is critical that the schema writer ensure all data can fit within the chosen offset range (e.g. buffer length $2^{\text{offset\_bits}-1}$). Offsets are stored wherever a variable-type field appears in a parent object. For example, in a struct or class definition, a member declared as string name; will be represented in the binary as a signed offset field occupying, say, 4 bytes (if offset_size=4). When reading, the code will add this offset to the base address of the offset field to find the string's length prefix. Because offsets are fixed-size, the in-place portion of any object (class or struct) remains a fixed length regardless of how many variable fields it contains (each variable field uses exactly offset_size bytes). This again ensures that fixed parts remain densely packed.

## 3.3 Type Categories

Below we summarize the type categories supported by NakedBytes and how they are represented:

- Primitive (Fixed) Types: bool (1 byte), signed/unsigned int8/16/32/64 (1–8 bytes), float32 (4 bytes IEEE-754), float64 (8 bytes IEEE-754). Each is stored in little-endian binary form. These have no special metadata, just the raw bytes in-place. (The system assumes IEEE and two's-complement[5].)
- Struct: A composite type with named fields (each field is any fixed or variable type). In the binary layout, a struct's in-place size is the sum of its fields' sizes (with padding for alignment as below). All of the struct's fields appear consecutively in the buffer. Since structs are fixed-size, they do not carry their own length; rather, the containing object must know how large the struct is (statically from the schema). If a field of a struct is variable-length, that field is represented by an offset within the struct. (Thus a struct may contain offset fields, but the struct size remains constant.)
- Fixed Array: Defined in schema as [T:n] where T is a type and n is a constant count. A fixed array is essentially like a struct: its elements (all of type T) are laid out back-to-back with no length prefix. This only applies to arrays of fixed-size types (scalars or structs); variable-size element types cannot be used in a fixed array, because then the total size wouldn't be known.
- Enum: A named enumeration, defined with an underlying integer type (e.g. int32). Enums are simply stored as that integer type. (We automatically generate integer constants, but at runtime it's just an int in little-endian.)
- Class (Nullable Struct with Compatibility): A top-level or member object that can have optional fields and versioning. Serialization of a class begins with a root length (an integer of the same size as an offset) that indicates how many bytes of fixed-in-place fields follow. Then, each declared field is output in declaration order. Fixed-size fields are written directly; variable fields are emitted as offset placeholders in-place, followed by their data appended later in the buffer. On reading, a class always treats the first value as its root length; if a reader's code only expects fewer bytes (old schema), it compares a field's intended offset with this length to see if the field exists. (If the offset exceeds the length, the field is considered absent.) This mechanism provides forward/backward compatibility. Conceptually, a class is like a FlatBuffers table, but without a separate vtable: it uses the root length to guard access. In schema, classes are declared similar to structs but with the keyword "class" and allow missing fields.
- Nullable Struct: This is a struct type that may be null. Represented simply as an offset in the parent. The offset is 0 to mean null (since a signed offset of 0 would point to itself, we agree to interpret 0 specially). If non-zero, it is treated as pointing to a struct of that type elsewhere. Unlike a full "class", a nullable struct has no versioning or root-length; it is just a convenient way to say "optional struct". In effect, the parent contains a fixed-size offset (0 = null, otherwise pointer) and the struct's bytes if present.

- Union: Defined as a choice of one among several types. Serialized as two fields of equal width (the offset-size): first an enum tag, then an offset. The tag is 0 for "none" and otherwise indicates which variant is stored (we treat the tag as an offset-width integer for simplicity). Then the offset points to the actual data of that variant. For instance, if a union can be an int32 or a struct Foo, the buffer will have a 4-byte tag followed by a 4-byte offset. If tag is 2, say, then the offset points to a Foo object. If tag is 0, no object is present (offset may be 0 as well). This matches FlatBuffers' union encoding[13].

- Vector: A variable-length homogeneous array (all elements same type). In the parent, a vector field is just an offset pointer. The pointed-to blob begins with an element count (an unsigned integer, same width as offset) followed by elements. If the element type is fixed-length, they are stored sequentially. If the element type is itself offsetable (like a string or nested object), then each element in the blob is actually an offset relative to the start of the vector blob. Different elements could even point to the same object (though usually unique). The vector offset always points to the count field.

- String: Stored as an offset (in the parent) to a blob with first a length (number of bytes) and then UTF-8 data. We also append a null terminator byte after the data (the null is not counted in the length). This is consistent with common string representations. The offset points to the length field[16].

- Blob: Raw binary data. Similar to string, but no character encoding; just a length prefix and that many bytes. Useful for e.g. embedded images. In schema, a blob may be declared as blob or something analogous.

## 3.4 Fixed vs Variable-Length Fields

As described, whether a field is stored in-place or by offset depends on its type:

- Fields of fixed-length types (primitive, enum, array, struct) occupy a fixed number of bytes in the containing object. Access to such a field is a direct dereference at the known offset. No checks or pointer chasing is needed. For example, in C you could compute uint32_t val = *(uint32_t*)(object_ptr + 8);

- Fields of variable-length (offsetable) types (nullable struct, class, union, vector, string, blob) occupy offset_size bytes in the container, which hold the signed offset to the actual data. To access these fields, code reads the offset and checks if it is non-zero (unless null is impossible, in which case the offset must be valid). If non-zero, it adds the offset to the address of the offset field (or the buffer base) to jump to the data. For example, a string field at position 16 with offset value 32 means the string data is at buffer address (buffer_base + 16 + 32). If the offset is 0 (or if a class's offset exceeds its root size), the field is treated as absent or default.

This model is very efficient: fixed fields are instant to access, and variable fields require exactly one addition and a pointer dereference (plus a null check). No loops or heap allocations are involved. In particular, accessing an element of a vector or string is done by reading the count/offset then indexing, which again is just pointer math. This zero-copy, pointer-based model is similar to FlatBuffers' access pattern[1].

### 3.5 Self-Referencing and Cyclic Structures

NakedBytes allows type-level cycles (circular references) by design, since offsets can point anywhere. For example, a struct Foo may contain an offset to a Bar, and Bar may contain an offset to a Foo. We handle this by forward-declaring types when generating code, as is done in FlatBuffers[17]: "circular references between types are allowed"[17]. However, object-level cycles (an actual cycle of pointer offsets at runtime) would require careful construction by the user and is not typical for static serialization. In normal use, NakedBytes structures form acyclic graphs of objects, like trees or DAGs, which can be serialized by simply writing each object once. In any case, our serialization algorithm (see below) can place objects in the buffer in any order because offsets are relative; it just needs to ensure each object's data is written before writing an offset to it. Overall, the offset mechanism inherently supports nesting and cross-references between objects.

## 4 Schema Definition Language

### 4.1 Overview

NakedBytes schemas are defined in a JSON-based format. The schema specifies the offset size, root type, version, and a list of type definitions. Each type can be one of: a primitive or enum definition, a union, or a structable type (struct, nullable struct, or class). This loosely resembles FlatBuffers' .fbs schema language, but in JSON. A schema might declare, for example, structs for fixed layouts, classes for versioned tables, unions for variants, and so on. Code generators use the schema to produce read/write accessors in the target language.

### 4.2 JSON Schema Structure

A complete schema is a JSON object with fields like:

Listing 1: Example NakedBytes Schema.

```json
{
    "offset_size": 4,
    "root_type": "MyClass",
    "version": 1,
    "types": [
        {
            "type": "struct",
            "name": "MyStruct",
            "members": [
                {
                    "name": "id",
                    "type": "string"
                },
                {
                    "name": "count_point_id",
                    "type": "string"
                },
                {
                    "name": "points",
                    "type": "vector[SunspecPointDef]"
                }
            ]
        },
        {
            "type": "class",
            "name": "MyClass",
            "members": [
                {
                    "name": "name",
                    "type": "string",
                    "default": "University of Ibadan"
                },
                {
                    "name": "school",
                    "type": "string"
                }
            ]
        },
        {
            "type": "union",
            "name": "MyUnion",
            "unions": [
                {
                    "name": "uint32",
                    "type": "uint32"
                },
                {
                    "name": "int32",
                    "type": "int32"
                },
                {
                    "name": "string",
                    "type": "string"
                }
            ]
        },
        {
            "type": "enum",
            "name": "Color",
            "base_type": "uint8",
            "enums": [
                {
                    "name": "Red",
                    "value": 0
                },
                {
                    "name": "Green",
                    "value": 1
                }
                ]
            }
    ]
}
```

- `offset_size`: (integer) number of bytes for all offsets (1,2,4, or 8). This choice fixes the maximum buffer size (e.g. 32-bit offsets allow up to $2^31$ bytes).
- `root_type`: (string) the name of the top-level type (must be a class or nullable struct) that serialization starts from.
- `version`: (int) a version ID for the schema (also stored in the buffer). Aids compatibility checks.
- `types`: an array of type definitions. Each type has a type field (enum, union, struct, class, or nullable struct) and a name. Structs/classes have a members list; unions have a cases list of type names; enums have a `base_type` and named values.

Members in a struct/class are objects with ”name”: ”...”, ”type”: ”T”, ”deprecated”: false, . The type can be a primitive, another struct/union/enum name, or array/vector notation like ”uint16[10]” or ”float[ ]” (the latter for a vector of floats). Arrays are denoted with [n] and vectors with []. A nullable struct type is declared by ”type”:”nullable_struct” in the JSON, or by appending a ? in some dialects. Fields may include an optional ”deprecated”: true flag to indicate they should no longer be generated in code.

## 4.3 Example Schema Snippet

For illustration, suppose we want a simple schema for a 2D point and a table of points:

Listing 2: Example NakedBytes Schema.

```
{
    "offset_size": 4,
    "root_type": "Polygon",
    "version": 1,
    "types": [
        {
            "type": "struct",
            "name": "Point",
            "members": [
                {
                    "name": "x",
                    "type": "float"
                },
                {
                    "name": "y",
                    "type": "float"
                }
            ]
        },
        {
            "type": "class",
            "name": "Polygon",
            "members": [
                {
                    "name": "id",
                    "type": "uint32"
                },
                {
                    "name": "vertices",
                    "type": "vector<Point>"
                },
                {
                    "name": "label",
                    "type": "string",
                    "deprecated": false
                }
            ]
        }
    ]
}
```

Here, Polygon is the root class. It has an inline uint32 id, an offset to a vector of Point, and an offset to a string label. A Point struct is two floats inline. A code generator would produce C++ classes (or other language structures) corresponding to these, with methods like get_id(), get_vertices(i), etc. (This is analogous to FlatBuffers codegen[9].)

## 4.4 Types, Members, Enums, Unions, Vectors

- Enums: Defined by base integer type and named values. Serialized exactly like that base type (e.g. a uint8 or int32). The JSON “values” map gives explicit integer assignments if needed, or auto-incremented by default.
- Unions: Defined with a list of possible types. For each union, an extra auto-generated enum tag is created. In our design the tag and offset fields in the buffer use the same width as the offset size, as noted.
- Structs / Nullable Struct / Class: Structs and nullable structs have no version tag, while classes include the root-length prefix. Nullable structs differ only in that they are stored as an offset (with 0=null) inside the parent.
- Vectors: Denoted by the vector¡type¿ notation. Equivalent syntax is type[]. In JSON, one can also write ”type”: ”TypeName[]”. The serializer will store the element count and then each element consecutively.
- Constraints: One constraint is that there must be no cyclic dependency in fixed-size nesting (to avoid infinite size); cycles are only allowed by indirection through offsets. The schema should also ensure the total size does not exceed the offset range.

### 4.5 Code Generation

A key part of the system is a schema compiler (flatc-like) that reads the JSON schema and emits code. For each struct/class, it generates a class/struct with methods to serialize into a buffer and deserialize (read) from a buffer. The deserialization API returns pointers or references into the original buffer. For each field, an accessor will either directly read the value (for fixed types) or compute an offset (for variable types). The code generation logic is similar to FlatBuffers' codegen[9]: it knows the layout so it emits direct memory offsets for each field. Deprecated fields are omitted from the generated code to avoid use of old fields. The result is a set of native classes that match the schema, making NakedBytes usage straightforward for the programmer.

## 5 Binary Representation

### 5.1 Canonical Layout

The serialized buffer has a canonical layout agreed by all implementations. It begins with a small header, then the root object's data, followed by any appended payload. In detail:

- Endian and Formats: All integer and float values use little-endian byte order, as is standard on modern CPUs[5]. Floats are IEEE-754. Signed integers are two's-complement. Booleans are 1 byte (0 or 1). This explicit specification guarantees cross-platform consistency. We assume (as most systems do) that the architecture uses these formats; if not, a platform-specific adapter must convert.
- Header: The very first field of the buffer is the total buffer size (including header), encoded as an unsigned integer of the chosen offset size (e.g. 4 bytes for 32-bit offsets). Immediately following is the schema version (also same width). These allow a reader to know the data length and schema version. (In some embedded use cases, one might omit the version in memory and rely on external versioning, but by default we include it.)
- Root Object: After the header, the root object's serialized form begins. If the root is a class, its first field is the "root length" (an integer, same width as offset) indicating how many bytes of its inline data follow. Then come the root's fields in declaration order. If the root is a struct or nullable struct, no length field is present (or null handling is used). Regardless, the layout of inline fields follows normal struct packing rules.
- Alignment: Each field is aligned to its natural boundary. Specifically, a type of size N is aligned to N bytes. Structs themselves are aligned to the largest alignment of any field they contain (again like C). Padding bytes (zero or undefined content) are inserted as needed to satisfy these alignments. The code generator may emit static asserts or warnings if heavy padding occurs. These alignment rules match FlatBuffers' guarantees for structs[12].

### 5.2 Offset Table and Pointers

There is no separate offset table or vtable block. Instead, offsets are embedded inline where needed. For a variable-length field at position p, the stored offset value is a signed integer that is added to p to get the actual data address. For example, if a vector's offset is at buffer index 10 and has value +20, the vector's data begins at index 30. If the offset were negative, it would point to an earlier position (this can occur if one serializes an object after its parent, then computes backward offsets). Because the offsets themselves take space (1, 2, 4, or 8 bytes each), the inline region of a class is larger if it has many variable fields. However, note that only offsets are stored inline – the actual data payloads are appended at the end.

## 5.3 Alignment Requirements

All primitive types are aligned to multiples of their size (1-byte types to 1, 2-byte to 2, etc.). Structs are aligned to their largest field. For example, a struct containing an int32 and a float64 will be aligned to 8 bytes. This is implemented by inserting padding bytes after fields if necessary. The buffer itself can be considered as a byte array, but the generated code will assume proper alignment. We follow the same scheme as FlatBuffers: structs explicitly declare alignment independent of the compiler's defaults, ensuring cross-platform consistency[12]. This avoids unpredictable padding differences.

## 5.4 Struct and Vector Layout

- Struct Layout: Each struct is serialized by concatenating its fields' binary representations, in the order declared. For fixed fields (like an int), the bits are placed in place. For offset fields (e.g. a nested string), the offset integer is placed where that field occurs. If the struct is nested inside another object, the struct's bytes are inlined. If the struct itself was pointed to by an offset, it is placed at the pointed-to location.

- Vector Layout: A vector is serialized as follows: first, a 32-bit (or offset-size) unsigned integer count of elements is written. (We choose 32 bits for the count to allow large vectors within a buffer.) Then the elements follow in sequence. If the element type is fixed (int, struct, etc.), those bytes come directly. If the element type is variable (e.g. string or another vector), then each element is stored as an offset relative to the start of the element. For example, a vector¡string¿ will have a count N, then N 32-bit offset values, followed by the actual string data blobs. FlatBuffers uses a similar scheme: vectors are "contiguous aligned scalar elements prefixed by a 32bit element count"[14], and pointers for variable elements.

## 5.5 String, Blob, and Bit Encoding

- Strings: In the final buffer, a string field is represented by an offset pointing to the string data. That data begins with a 32-bit length (the number of bytes) followed by that many UTF-8 bytes and a trailing null. The length does not count the null. For example, the string "hi" would be serialized as bytes [2 0 0 0 'h' 'i' '
0'] somewhere in the buffer. The offset in the parent points to the first length byte. This matches the flatbuffers notion that "Strings are simply a vector of bytes, and are always null-terminated"[16].

- Blobs: Similar to strings, a blob is preceded by a 32-bit length and followed by raw bytes. No null terminator is needed. The parent holds an offset to the length field. Blobs are useful for arbitrary binary data.

- Bitsets / Optional Flags: We do not natively support an explicit bitset type. Optionality is handled by offsets (null=0) or by class length checks. There is no compact "presence bitmap" stored.

### 5.6 Handling Null and Default Values

A field is considered null or absent if its offset is zero (or out-of-bounds for class fields). In practice, for nullable struct, class, and union, the default offset of 0 means "no data". For example, a string with a 0 offset is an empty string or null string depending on interpretation. For unions, an enum tag of 0 (NONE) indicates no object, and the offset may be ignored. Default values for primitive types behave as in C/C++: if a field is absent, the reader will return 0 for integers/floats, false for bool, and the first enum value for enums. (The generated accessor uses a constant default, e.g. 0.) This is analogous to FlatBuffers: when a field is not present (vtable entry 0 or index out-of-range), it returns the default value[11]. In NakedBytes, since we have no vtable, an absent field means the offset is null, so the accessor can detect this and return the default. For fixed structs, all fields must exist so no default logic is needed. For class fields beyond the root length, we treat them as absent and use defaults.

## 6 Serialization Process

### 6.1 Arena Model

Serialization into NakedBytes is done in a single contiguous buffer (the arena). The process begins by allocating (or receiving) a buffer of appropriate maximum size. We then reserve the first few bytes for the header (length and version). Next, we serialize the root object. Any fixed-size fields (structs, ints, etc.) are written directly into the buffer at the current write offset (which starts just after the header). For variable-length fields, we do one of two things: if the field's value is itself a nested object, we later serialize that object and note its position; if the field is a scalar list or string, we similarly serialize that data later. Conceptually, we perform a two-phase serialization: write placeholders for offsets, then fill in data. Concretely, one can do this by writing out the inline (fixed) portion of each object in a top-down order, while pushing all payloads (strings, vectors, nested objects) into a queue. After all inline fields are written, we append each payload to the end of the buffer one by one, updating the previously written offsets. Because offsets are relative, we add the offset as (payload_position – offset_field_position). This approach avoids any extra allocations: everything goes into the arena. (FlatBuffers takes a similar approach by building buffers "backwards" to minimize bookkeeping[18], but our method is forward; both achieve one-pass construction.)

### 6.2 Construction of Objects

When serializing a struct or class instance, we follow its declaration order: 1. For each fixed-size field, simply encode its value (and include padding for alignment as needed). 2. For each variable-length field, record the current buffer position for where the offset should go, temporarily write a placeholder (e.g. 0). Also enqueue the field's actual data (string bytes, sub-object, vector content) for later emission. When all fields of the object are processed, we move on. After finishing the root, we process the queue of deferred data. For each deferred item, we note the current end of buffer, write the item's data (e.g. string length+chars, or vector elements, or recursively serializing a nested object), and then go back and fill in the offset in the parent's placeholder. The offset value is computed as the signed difference from the parent's offset-field location to the new data's location. This completes the link. Classes also include writing the root-length at the very beginning of their inline data. The root-length is simply the size in bytes of all fixed fields (plus padding) of that class. This is recorded before writing the fields, so that it can be checked on read.

### 6.3 Offset Fixups

After all inline fields are written, each variable field must have its offset updated. Because we reserve space for the offset early, when the payload is written we know the target address. We subtract the position of the offset field itself (plus any size where needed) to get the relative offset. Because we use signed offsets, negative values would appear if for some reason payload came before parent, but normally all payloads are appended after, so offsets are positive. In either case, we ensure to write the correct offset value in the reserved spot. This is a constant-time calculation per field. No data is moved after this point. subsectionValidation During serialization, a final check ensures that the total buffer length fits within the chosen offset range. For example, if offset_size=1 (max 127), we must verify the buffer is not longer than 127 bytes. If it is, we would either fail or require re-serialization with a larger offset type. Additionally, when constructing objects, the code generator may assert that offsets do not overflow. After serialization is complete, the header is finalized by writing the total length (and version) at the start. The buffer is then ready to use.

## 7 Deserialization and Data Access

### 7.1 Zero-Copy Traversal

To read data, NakedBytes makes use of simple pointer arithmetic. A read-only pointer (or byte pointer) is taken to the start of the buffer (plus header). To get a field value, the generated accessor code does one of the following: - Fixed field: Directly interpret the bytes at the known offset. e.g. *((int32_t*)(base + 8)). - Offset field: Read the stored offset integer, check for null (0) if needed, then compute a new pointer: target = base + offset. For example, char* s = (char*)(base + *(int32_t*)(base + 12)); would access a string. The pointer math is constant-time. This is exactly how formats like FlatBuffers provide "random access" to fields[1]. The only operations involved are integer addition and pointer dereference. There is no loop or recursion unless the user code walks a structure (e.g. iterating through a vector). Thus access is extremely fast – essentially the cost of following a C pointer. This matches the stated goal: "the only operations used are pointer addition and dereferencing" for primitives and nested structs.

### 7.2 Pointer Reconstruction via Relative Offsets

Because offsets are relative, code can easily convert them to absolute pointers in the buffer. In practice, the generated code often computes an address on-the-fly and returns a typed pointer or reference. For example, given a buffer base buf, if an offset field ofs is read at address (buf

+ 16), then the code computes buf + 16 + ofs_value. This pointer points to the actual data (length prefix or value). For instance, accessing a vector's elements might look like:

```
uint8_t* vec_start = buf + vec_offset_addr +
                   read_u32(buf+vec_offset_addr);
uint32_t length = *(uint32_t*)vec_start;
for (uint32_t i = 0; i < length; i++) {
    // access element at vec_start + 4 + i*sizeof(element)
}
```

Because the offset encoding is consistent, this pointer reconstruction is reliable across all languages.

### 7.3 Safety and Bounds Checking

Generated accessors include basic safety checks to avoid out-of-bounds reads. For example, a class accessor will check if the requested field index is within the recorded root-length. If a field is beyond the serialized length, the accessor returns a default or null. For vectors and strings, the code ensures not to read beyond the buffer end (since the total length is known). In FlatBuffers, a vtable lookup out-of-range means "field not present, use default"[11]; similarly, NakedBytes treats offset==0 or out-of-range as absence. In debug builds, one could also insert explicit bounds checks (e.g. verifying that pointer + data_size ¡= total_length) to prevent corrupted data access. subsectionRandom Access Characteristics Because data is located via offsets, any field or element can be accessed in O(1) time given its offset in the schema. There is no need to scan or parse previous fields. This enables true random access. For example, you can jump to the N-th element of a vector without iterating from the start, because the vector has a count and fixed stride (for fixed-size elements) or stored offsets (for variable elements). Even nested fields require only a fixed sequence of pointer dereferences. This is in contrast to parsed formats (e.g. JSON) where accessing a deep field may require scanning the entire prefix. The zero-copy nature allows extremely fast point queries in the buffer, which is ideal for high-performance use cases. In short, NakedBytes delivers the same kind of "immediate structured access" that has made FlatBuffers popular in real-time systems[1][3].

## 8 Example

This section provides a concrete example of serializing and deserializing a simple object using NakedBytes.

### 8.1 Schema Definition

Below is a sample NakedBytes schema in JSON format that defines a Packet structure containing various fields, including a union type, string, and nested structs.:

Listing 3: A simple NakedBytes Schema for defining a Packet.

```
{
    "offset_size": 2,
    "version": 5,
    "root_type": "Packet",
    "types": [
        {
            "name": "Monster",
            "type": "struct_offset",
            "members": [
                {
                    "name": "name",
                    "type": "string"
                }
            ]
        },
        {
            "name": "Weapon",
            "type": "struct",
            "members": [
                {
                    "name": "name",
                    "type": "string"
                },
                {
                    "name": "damage",
                    "type": "uint32"
                }
            ]
        },
        {
            "name": "AnyPower",
            "type": "union",
            "unions": [
                {
                    "name": "Monster"
                },
                {
                    "name": "Weapon"
                }
            ]
        },
        {
            "name": "Packet",
            "type": "struct",
            "members": [
                {
                    "name": "id",
                    "type": "int16"
                },
                {
                    "name": "description",
                    "type": "string"
                },
                {
                        "name": "power",
                        "type": "AnyPower"
                    },
                    {
                        "name": "length",
                        "type": "uint32"
                    },
                    {
                        "name": "you",
                        "type": "Monster"
                    }
                ]
            }
    ]
}
```

## 8.2 Serialization Example

To show a serailized example, consider the following Packet object: Using the above schema, we can serialize a Packet object as follows:

Listing 4: A simple NakedBytes Schema for defining a Packet.

```
{
    "id": 1,
    "description": "A sample packet!",
    "length": 10,
    "power_type": "Weapon",
    "power": {
        "name": "Most dangerous weapon!!!",
        "damage": "0xdeadadde"
    },
    "you": {
        "name": "Humans!!!"
    }
}
```

After serialization, the resulting binary buffer (in hexadecimal) dump by the xxd program:

Listing 5: Serialized NakedBytes Buffer for the Packet object.

```
00000000: 5a00 0500 0100 0e00 0100 1e00 0a00 0000  Z...............
00000010: 3c00 a5a5 1000 4120 7361 6d70 6c65 2070  <.....A sample p
00000020: 6163 6b65 7421 0000 0800 a5a5 dead adde  acket!..........
00000030: 1800 4d6f 7374 2064 616e 6765 726f 7573  ..Most dangerous
00000040: 2077 6561 706f 6e21 2121 0000 0200 0900   weapon!!!......
00000050: 4875 6d61 6e73 2121 2100                 Humans!!!.
```

Breaking down the buffer: Since the offset size specified in the schema is 2 bytes, the maximum is 32767 since offset can be in both directions and all offsets, union type enums, length and version are 2-byte integers and the value is in little-endian format.

- The first 2 bytes (5a00) represent the total buffer length (90 bytes).

- The next 2 bytes (0500) represent the schema version (5).
- The next 2 bytes (0100) represent the Packet.id field (1).
- The next 2 bytes (0e00) represent the offset to the description string (14 bytes ahead).
- The next 2 bytes (0100) represent the union type to the power which is 1 which represents the weapon type.
- The next 2 bytes (1e00) represent the offset to the power object (30 bytes ahead).
- The next 2 bytes (0a00 0000) represent the Packet.length field (10).
- The next 2 bytes (3c00) represent the offset to the Packet.you which is of Monster object which is a struct_offst type(60 bytes ahead).
- The next 2 bytes(a5a5) represent the padding added to meet the alignment requirements for the Packet object.
- The next bytes represents the description string starting with its two bytes length prefix (0a00) followed by the UTF-8 bytes for "A sample packet!" and a null terminator (00).
- The next 2 bytes (0800) represent the offset of the weapon.name string (8 bytes ahead).
- The next 2 bytes (a5a5) represent the padding added to meet the internal alignment requirements for the uint32 damage field in Weapon object.
- The next 4 bytes (deadadde) represent the Weapon.damage field (0xdeadadde).
- The next bytes represent the weapon.name string starting with its two bytes length prefix (1800) followed by the UTF-8 bytes for "Most dangerous weapon!!!" and a null terminator (00).
- The next 1 byte (00) represent the padding added to meet the alignment requirements for the Monster object.
- The monster object you offset ahead start here with the next 2 bytes (0200) represent the offset of the Monster.name string (2 bytes ahead).
- The next bytes represent the Monster.name string starting with its two bytes length prefix (0900) followed by the UTF-8 bytes for "Humans!!!" and a null terminator (00).

## 8.3 Deserialization Example

The deserialization process is actually a misnomer, since there is no deserialization being done but rather just accessing the data in-place via pointers dereferencing and offset calculations. Below is an pseudo code example of how one would access the fields of the Packet object from the serialized NakedBytes buffer. To access the id field:

Listing 6: Accessing the Packet.id field from the NakedBytes buffer.

```
int16_t packet_id = *(int16_t *)(buffer + 4); // offset 4 bytes from start of buffer to skip the
    header
```

To access the description string:

Listing 7: Accessing the Packet.description string from the NakedBytes buffer.

```
int16_t desc_offset = *(int16_t *)(buffer + 6); // offset to description field
char *description = (char *)(buffer + 6 + desc_offset); // compute absolute address
uint16_t desc_length = *(uint16_t *)description;
char *desc_str = description + 2; // skip length prefix
```

## 8.4 c++ Code Generation

Below is an example of the C++ struct that would be generated by a NakedBytes code generator for the Packet object defined in the schema. This struct includes methods to access each field in the buffer using pointer arithmetic and offset calculations.

Listing 8: Deserializing a NakedBytes Buffer into a Packet object.

```
struct PacketRoot
{

#define PACKET_ID_OFFSET 0
#define PACKET_DESCRIPTION_OFFSET 2
#define PACKET_POWER_TYPE_OFFSET 4
#define PACKET_POWER_OFFSET 6
#define PACKET_LENGTH_OFFSET 8
#define PACKET_YOU_OFFSET 12
#define PACKET_PAD14_OFFSET 14
#define PACKET_ALIGNMENT 4
#define PACKET_SIZE 16

    const int16_t id() const
    {
        return *reinterpret_cast<const int16_t *>(&data_[PACKET_ID_OFFSET + 2 *
            OFFSET_SIZE]);
    }

    const Offset<String> &description() const
    {
        const int16_t offset = PACKET_DESCRIPTION_OFFSET + 2 * OFFSET_SIZE;
        return *reinterpret_cast<const Offset<String> *>(&data_[offset]);
    }

    const AnyPower_enum power_type() const
    {
        return *reinterpret_cast<const AnyPower_enum *>(&data_[
            PACKET_POWER_TYPE_OFFSET + 2 * OFFSET_SIZE]);
    }
    const AnyPower &power() const
    {
        const int16_t offset = PACKET_POWER_OFFSET - 2 + 2 * OFFSET_SIZE;
        return *reinterpret_cast<const AnyPower *>(&data_[offset]);
    }

    const uint32_t length() const
    {
        return *reinterpret_cast<const uint32_t *>(&data_[
            PACKET_LENGTH_OFFSET + 2 * OFFSET_SIZE]);
    }

    const Offset<Monster> &you() const
    {
        const int16_t offset = PACKET_YOU_OFFSET + 2 * OFFSET_SIZE;
        return *reinterpret_cast<const Offset<Monster> *>(&data_[offset]);
    }

    static constexpr size_t nakedbytes_sizeof = 16;

private:
    PacketRoot() = delete;
    PacketRoot(const PacketRoot &) = delete;
    PacketRoot &operator=(const PacketRoot &) = delete;
    unsigned char data_[1];
};
```

## 9 Technical Specification Summary

- Primitive types supported: uint8, uint16, uint32, uint64, int8, int16, int32, int64, float32 (float), float64 (double), bool, Enums (user-defined int types). All are stored in little-endian as raw bytes (fixed size).
- Offset size: Configurable 1, 2, 4, or 8 bytes (unsigned or signed accordingly). The schema's offset_size field must match the architecture ($maximumoffset < 2^{(}offset\_size * 8 - 1)$ for signed). We suggest 4 bytes (32-bit) by default. The choice determines buffer max size. (FlatBuffers' default is 32-bit to stay cross-platform[15].)
- Size limits: The total buffer length (and thus any object's offset) must fit within the signed range of the offset (e.g. ¡ $2^3$1 for 32-bit). Each class's root length and each vector length are also limited by offset capacity (e.g. vector element count fits in 32 bits).
- Endianness: Little-endian for all multi-byte values. This matches the prevalent architecture requirement[5]. Serialization always writes in little-endian; on a big-endian machine the code generator would need to include byte-swap conversions.

- Versioning fields: The first two fields in the buffer (each of width offset_size) are the total byte length and the schema version. These can be used to identify the data or check compatibility.
- Reserved Fields: In the class definition, fields can be marked "deprecated" so they generate no code. Future work may define reserved offset patterns for extensions, but currently any new fields should be appended by increasing schema version.
- Alignment/Padding: Fields are naturally aligned as above; extra padding is inserted as needed. The code generator reports the static size of each struct/class (including padding).
- Floating-Point: IEEE-754 binary32 and binary64. Stored with no transformation (just little-endian)[5].
- Null Representation: For any offset field, a 0 value means null/absent. For unions, an enum tag of 0 (NONE) means no object; in that case the offset should also be 0.
- Forward/Backward compatibility: Classes include a "root size" to check if a requested field exists. If field_offset ¿ root_size, the field is considered absent (backward compatibility). New fields should only be appended (forward compatibility)[10].
- Implicit Invariants: The schema must not contain contradictory definitions. For example, an array field must have a fixed count, a vector must specify element type, and cycles must be via indirection. The serializer assumes the schema is correct.

## 10 Conclusion and Future Work

NakedBytes provides a unified, offset-based serialization framework optimized for speed and compactness. By storing data in a canonical in-memory format with only relative offsets for variable parts, it achieves near-zero-copy access[1] and minimal overhead[3]. The design draws on lessons from FlatBuffers and Cap'n Proto but streamlines them: it drops vtables to save space, supports schema evolution, and enforces alignment for reliability. The JSON schema and code generation make it adaptable to many languages, and the arena model means it works in the tightest embedded environments. Potential future enhancements include: bit-packing of small fields, optional compression of large blobs, richer default-value support, and tools for schema migration. We also plan to expand the language bindings (e.g. Python, Java) and investigate formal verification of backward compatibility rules. Ultimately, NakedBytes aims to serve as a drop-in replacement for less efficient serializers in domains from IoT to real-time analytics, giving developers both the familiarity of C-like types and the performance of direct memory access.