

Source: b3u docs/why_not_protobuf.md @ d89933f — curated 2026-09-01; edits land in the source first.

Why Not Protobuf?

b3u.dev and schema-first serialization, compared honestly

Draft 0.1 — 2026-09-01

Every developer who hears "define the data model once, get matched parsers in five languages" asks the same question within thirty seconds: "why wouldn't I just use protobuf?" It is the right question, and it deserves a direct answer rather than a pitch. This page is that answer. It is deliberately short.

1. The short answer

Protocol Buffers solves the problem of **inventing a new wire format** for data you control end to end. b3u.dev solves the problem of **reading, byte-identically, a format that already exists** — an HL7 clinical message, an X12 EDI envelope, a COBOL copybook record, a protocol payload, your own configuration DSL — in every language your stack uses, without changing a byte of the data or asking anyone upstream to adopt anything.

If your data is greenfield, flows only between services you own, and gRPC fits your architecture: use protobuf. It is mature, fast, and superbly supported. This page exists for the cases where that sentence is not true — which, in integration-heavy domains, is most of them.

2. The structural difference

A `.proto` file describes **messages**: named fields with types, encoded in protobuf's own binary wire format. It cannot describe someone else's syntax. There is no `.proto` for an HL7 v2 message, an X12 interchange, a copybook record, or a domain-specific language — those formats have grammar (delimiters, nesting, positional structure, context) that a field list cannot express.

b3u.dev is built on a compiler-compiler: the definition you write is a **grammar**, and what you get back is a set of generated parsers — in C++, Java, Python, JavaScript, and Rust — for the format

as it is. The same definition also yields a compact binary form of every parsed document and a decompiler that reproduces the original text **byte for byte** in each runtime. That byte-identity is the verification story: five independent runtimes, one canonical output, checked with `diff` rather than trusted by assertion. It has been demonstrated publicly on HL7 ORU^R01, X12 envelopes, COBOL copybook data, and a multisig custody model (the four reports at b3u.dev/docs).

Protobuf, by design, offers no such property: its serialization is explicitly documented as non-canonical — the same message may encode to different bytes across languages and library versions, which is why deterministic output is an opt-in, per-implementation feature and byte-comparison across stacks is not part of its contract.

3. Side by side

	Protocol Buffers	b3u.dev (CCS)
Definition unit	message (field list)	grammar (full syntax)
Wire format	its own (all parties must adopt it)	the format you already have; plus an optional fast binary
Existing formats (HL7, X12, copybooks, DSLs)	out of scope	the core use case
Cross-language agreement	semantic (values match)	byte-identical canonical
Verification story	encoding non-canonical	output in all five runtimes
Field-level protection	shared test vectors	diff of decompiled bytes
	external to the format	built in: per-record ECIES/AES-GCM blobs, wrong-key = hard refusal
Binary load performance	excellent	published: ~2× simdjson DOM on JSON; 20–30× faster than the text form
Languages	10+ incl. Go, C#	C++, Java, Python, JavaScript, Rust
RPC framework	gRPC, first-class	none – data layer only
Maturity / ecosystem	~2 decades, vast	young; four public demonstrations, seeking design partners

4. When to use which

Use **protobuf** when you are defining a new internal API between services you control, you want gRPC, you need Go or C#, or your organization has already standardized on it. Nothing on this

page argues otherwise.

Look at b3u.dev when any of these is true:

- The format **already exists** and you cannot change it: a standard (HL7, X12), a legacy layout (copybooks), a protocol payload, a partner's file format, your own DSL.
- The same format is **hand-decoded in two or more languages today** — a Python backend and a TypeScript frontend, a Java integration and a C++ engine — and the copies drift.
- You must **prove** the implementations agree, not assume it: compliance, audit, consensus-critical or safety-critical readers. Byte-identical canonical output turns that proof into a `diff`.
- Records need **field-level confidentiality at rest** with per-record keys and hard refusal on the wrong key, as part of the data layer rather than bolted on around it.

The two also compose: nothing prevents protobuf RPC between services whose *payloads in foreign formats* are parsed and verified by generated CCS readers.

5. What is not claimed

- No Go and no C# today — if your critical path is Go, protobuf (or hand-written decoders) remains your answer; the gap is stated in the web3 report and is a known open item.
- No RPC framework, no service mesh, no IDL for APIs — b3u.dev is a data-layer engine, not a gRPC alternative.
- No claim of protobuf-beating encode/decode microbenchmarks; the published numbers are load-time comparisons against JSON parsers (simdjson, nlohmann) on public corpora, stated in the reports.
- Protobuf's ecosystem, tooling, and two decades of production hardening are real advantages and are not disputed anywhere on this page.

6. The offer

The standing offer from the other reports applies here unchanged: send one sample file of a format you maintain in two or more languages, and we return the five-runtime demonstration on **your** data — parsers, byte-identity check, and the encrypted-record form — free, within a week.

Contact: support@b3u.dev.

Everything cited above is published in two places: **b3u.dev/docs** holds the reports and papers; **b3u.dev/usecases** holds the runnable use cases behind them. The short map:

- **The four vertical demonstrations** — HL7 clinical messages (b3u.dev/docs/ccs_java_hl7.pdf), X12 EDI envelopes (b3u.dev/docs/ccs_x12.pdf), COBOL copybook data (b3u.dev/docs/ccs_cobol_data.pdf), multisig custody for web3 (b3u.dev/docs/ccs_web3.pdf).
- **Benchmarks** — the JSON load numbers against nlohmann and simdjson (b3u.dev/docs/CCS_Compiler_Compiler_arXiv_Draft_0.1.pdf; runnable at b3u.dev/usecases/ccs_json_bench) and the binary-vs-text crossover study (b3u.dev/docs/MeTTa_Bulk_Load_Benchmark_Draft_0.1.pdf; runnable at b3u.dev/usecases/metta_bench and /usecases/metta_bench_rust).
- **MeTTa language support** — a full language, not just a data format, on the same engine (b3u.dev/docs/MeTTa_on_CCS_Draft_0.1.pdf; b3u.dev/usecases/metta).
- **Cross-language translation** — the Python-to-TypeScript RealWorld case study (b3u.dev/docs/Cross_Language_Translation_arXiv_Draft_0.1.pdf; b3u.dev/usecases/p2t).
- **The shell** — invoking the target compiler from each of the five languages and doing SCB processing inside the same process, no subprocess round-trip (b3u.dev/usecases/ccs_shell).
- **Field-level protection** — the obfuscation layer and the position paper behind it (b3u.dev/docs/A_Second_Kind_of_Secret_Draft_0.1.pdf; b3u.dev/usecases/obfuscator).