Wire Formats: Implementation Tradeoffs

Brendan Sechter

2026-01-28

The companion article Wire Formats, What They Are established that data interchange encodings, protocol framing, and instruction encodings are the same kind of artifact, and closed by naming five questions that every wire format must answer. This article takes those questions, adds six more that emerge once a format has to survive contact with time, with tooling, and with adversaries, and treats each as a tradeoff rather than a problem with a correct answer. The families are not separated here. Each section shows how one tradeoff appears in all three, because the point of the pairing is that the decisions transfer.

A tradeoff in this sense is a choice where improving one property necessarily degrades another. Where a design appears to escape a tradeoff, it has usually moved the cost somewhere less visible rather than eliminated it, and part of the work below is saying where the cost went.

Where the Schema Lives

The schema is the agreement about what the bytes mean. It can travel inline with every message, be distributed out of band and referenced, or be fixed permanently by a published standard.

Inline schemas make a message self-contained. Any reader can decode it with no prior arrangement, which is why JavaScript Object Notation succeeded far beyond the cases its designers anticipated. The cost is paid on every single message, forever, for information that almost never changes.

Out-of-band schemas move that cost to a one-time distribution step. Apache Avro writes the schema once in a file header and then emits values with almost no per-record metadata, which is close to optimal for large homogeneous batches and poor for a single small message to an unknown party. Header compression in HTTP/2 is the same idea applied to a live conversation, where the HPACK dynamic table is a schema that both ends build cooperatively as they talk. For a header field of full size H that the table can render as an index of size I, a table hit rate θ gives an effective size

$$\bar{H} = \theta I + (1 - \theta)\, H$$

so a table hitting nine times in ten reduces a fifty-byte header to under six bytes when the index costs one. The gain comes entirely from repetition, which is why the technique helps a long conversation and does nothing for a single request.

Fixed schemas are the instruction-encoding position. A processor does not receive a description of its instruction set, because the set is fixed by the architecture specification and burned into the decoder. This gives the lowest possible overhead and the least possible flexibility, and it is why adding an instruction is a multi-year undertaking rather than a deployment.

The cost structure is easy to state. For N messages where the schema costs s bytes and each message carries d bytes of data plus m bytes of inline metadata, the inline total is

$$T_{\text{inline}} = N\,(d + m)$$

and the out-of-band total is

$$T_{\text{external}} = s + Nd$$

so distributing the schema separately wins when

$$N > \frac{s}{m}$$

The threshold is a ratio, not a constant. A format with heavy inline metadata reaches it after a handful of messages. A format with one-byte field tags may never reach it for a low-volume protocol, which is why Protocol Buffers keeping small tags inline is a reasonable middle position rather than a failure to commit.

Finding the End of a Unit

Every reader must know where a unit stops. Length prefixing, delimiting, and fixed width are the three answers, and each fails differently.

Length prefixing lets a reader allocate exactly once and detect truncation immediately, and it makes the length an attacker-controlled input. A declared length that the sender never fulfils is a resource-exhaustion vector. A reader must therefore enforce

$$\ell \le \ell_{\max}$$

on the declared length ℓ before allocating, where $\ell_{\max}$ is a bound the reader chooses independently of the sender. Checking after allocation is not a check.

Delimiting requires no lookahead and permits streaming without knowing the total size in advance, and it forces an escaping scheme for payloads that contain the delimiter. Escaping is where subtle bugs live, because encoder and decoder must agree exactly on what is escaped and the failure mode is silent corruption rather than a clean error. Escaping also costs size. If the delimiter occurs in payload with probability q per byte and each occurrence expands to two bytes, a payload of p bytes becomes

$$p' = p\,(1 + q)$$

in expectation, so a delimiter chosen to be rare in the expected data costs almost nothing and one chosen carelessly approaches a doubling.

Fixed width removes the question entirely at the cost of flexibility. The base RISC-V integer instruction set fixes instructions at thirty-two bits, so the address of the next instruction is always known without decoding the current one, which is what makes wide superscalar fetch practical. The compressed extension reintroduces variable width for density and pays for it with a more complex decoder.

A decoder constrained to a single pass cannot look ahead, which turns boundary discovery from a convenience into a hard constraint. The compiler literature treats the same problem as the forward reference, where a jump target is not yet known at the moment the jump must be emitted, and the standard resolution is to write a placeholder and patch it once the target

resolves. Fixup Tables and the Forward-Jump Problem develops that mechanism, and Block-Structured Control Flow and Single-Pass Validation shows how a format can be shaped so the problem does not arise. A streaming decoder faces the identical choice between buffering until the extent is known and emitting provisional structure that a later byte corrects.

Instruction encodings show the tradeoff at its sharpest because the decoder is silicon. A variable-width encoding means the processor cannot know where instruction $j + 1$ begins until it has partly decoded instruction j, which serialises a step that the designer very much wants parallel.

The Unrecognised

A reader will eventually encounter something it does not understand. What it does then is a design decision with consequences that outlast every other choice in the format.

The permissive position skips the unknown and continues. Protocol Buffers achieves this through the wire type in each field tag, which tells a decoder how many bytes to skip without knowing what the field means. This is what makes it possible to add a field to a message that old readers will tolerate, and it is the single most important property for a format used across independently deployed systems.

The strict position rejects anything unrecognised. This is correct where the reader must fully understand a message to act safely on it, and it is the right default in consensus systems, where a participant that silently ignores part of a transaction may compute a different result from its peers and cause a chain split rather than a local bug.

The permissive position is the robustness principle in its classical form, which counsels being liberal in what a reader accepts. That advice has been substantially revised. The language-theoretic security position holds that a parser should recognise exactly the language the specification defines and nothing beyond it, and that the discipline of building parsers from an explicit grammar rather than from ad hoc control flow eliminates whole categories of defect, an argument Anantharaman and colleagues develop with worked construction rather than exhortation. Sassaman, Patterson and Bratus argue that liberal acceptance is precisely what creates parser differentials, since two readers that are liberal in different ways disagree about what a message means, and RFC 9413 restates the modern position that tolerance defers cost rather than removing it and that divergent implementations become a maintenance and security burden.

The two positions produce opposite failure modes. Permissive readers stay compatible and can act on messages they only partly understand. Strict readers refuse to act on incomplete understanding and require coordinated upgrades.

Schema evolution formalises this. Let W be the writer schema and R the reader schema. Write dec_R for the reader's decode function and enc_W for the writer's encode function, and let $\preceq$ order schema versions. Backward compatibility is the requirement

$$\mathrm{dec}_R\left(\mathrm{enc}_W(v)\right) = v \quad \text{for all } W \preceq R$$

and forward compatibility is the same condition for all $W \succeq R$. A format supports backward compatibility when a reader at R can read data written at an older W, and forward compatibility when a reader at R can read data written at a newer W. Avro resolves the two schemas explicitly at read time, as described in its specification, which makes the compatibility rules a stated part of the format rather than an emergent property. Protocol Buffers obtains forward compatibility from skippable unknown fields and constrains what changes are safe, and its proto3 field presence rules document where the constraints bite.

The rule that follows applies to all three families. A format cannot be simultaneously strict about the unrecognised and tolerant of independent deployment. Choosing strictness is choosing coordinated upgrades.

Paying for Inspectability

A format that a human can read with ordinary tools is easier to debug, easier to log usefully, and easier to learn. That property costs bytes and cycles on every message.

The argument for paying is operational rather than technical. When a distributed system misbehaves at three in the morning, a text format can be read directly from a capture, and a binary one requires tooling that must itself be correct and available. A substantial part of the reason JSON displaced more efficient predecessors is that the debugging story was better, and debugging cost is real cost.

The argument against is that the price is paid continuously by every message in production to benefit the rare occasion when a human looks. HTTP/2 moved to binary framing precisely because the text framing of version 1.1 could not express stream multiplexing without ambiguity, and the human-readability of the header block was not worth the constraint.

A common resolution is to keep the format binary and invest in tooling that renders it readable on demand, which converts a per-message cost into a one-time engineering cost. This works when the tooling is reliable and available at the moment of need, and fails exactly when it is not, which tends to be during the incidents where it matters most.

Canonical Encoding

Canonical encoding requires that a given value have exactly one valid byte representation. Let $E(v)$ be the set of byte sequences a format permits for the value v. Canonicity is the condition

$$\forall v, \quad |E(v)| = 1$$

which is strictly stronger than the unambiguity every wire format already requires. Unambiguity says each byte sequence denotes one value. Canonicity says each value has one byte sequence. A format can satisfy the first and violate the second, and most do. Most formats do not require this, because it constrains encoders for no benefit in ordinary use.

It becomes mandatory the moment bytes are hashed, signed, or compared. If two encoders may legitimately produce different bytes for the same value, then a signature over those bytes verifies the encoding rather than the value, and a hash cannot serve as an identity. CBOR addresses this with the deterministic encoding requirements in RFC 8949, and ASN.1 has carried the distinction since its Distinguished Encoding Rules in ITU-T X.690, where Basic Encoding Rules permit choices that Distinguished Encoding Rules remove.

The sources of non-canonicity are consistent across families. Integers may admit multiple lengths, as a varint that could be written with fewer bytes but is not. Map or field ordering may be unconstrained. Optional fields may be present with a default value or absent. Floating point may admit multiple representations of the same quantity. Padding may be unconstrained.

Consensus systems have no choice. Every participant must derive the same result from the same transaction, so the encoding must be canonical, non-canonical encodings must be rejected rather than normalised, and the rejection must itself be part of the specification. Silently normalising is worse than rejecting, because two implementations may normalise differently.

The cost is encoder freedom. A canonical format cannot let an encoder choose a faster representation, and it cannot add a representation later without a version change.

Compactness Against Decode Cost

Making bytes smaller usually makes them more expensive to interpret, because compactness comes from removing redundancy and redundancy is what makes decoding cheap.

Varint encoding is the clearest case. It shortens small integers and requires a per-byte loop with a data-dependent branch, where a fixed-width integer is a single aligned load. Zero-copy formats invert the tradeoff by spending bytes on alignment and offsets so that reading becomes pointer arithmetic, and Cap'n Proto is explicit that it is trading size for the elimination of a decode pass.

The choice depends on which resource binds. For a message of size S crossing a link of bandwidth B and then being decoded at rate D bytes per second, the total time is approximately

$$t = \frac{S}{B} + \frac{S}{D}$$

Compression that reduces S to αS for $\alpha < 1$ while reducing the effective decode rate to βD for $\beta < 1$ helps when

$$\frac{\alpha}{B} + \frac{\alpha}{\beta D} < \frac{1}{B} + \frac{1}{D}$$

On a slow link the bandwidth term dominates and compactness wins. Within a data centre, where B is large, the decode term dominates and the same choice loses. This is why formats that win on mobile networks lose on local interconnects, and why the correct answer changes with deployment rather than being a property of the format.

Where the decoder rather than the link is the constrained resource, the calculation shifts again. Symbol Tables, Scope Popping, and Bounded Working Memory treats the compiler case in which the binding constraint is the working set a decoder must hold rather than the size of what it consumes, which is the situation for any device decoding a stream larger than its memory.

Instruction encodings face the identical calculation with different units. A compressed encoding reduces instruction memory and increases decoder complexity, and whether that is worth it depends on whether the design is constrained by memory or by decode throughput.

Alignment, Endianness, and the Cost of Portability

Byte order and alignment are the oldest portability hazards in the field and the ones most often assumed away.

A format that fixes byte order imposes a conversion on machines of the opposite convention. A format that carries a byte-order marker avoids the conversion and requires every reader to handle both, which doubles the paths through the decoder and therefore doubles what must be tested. Network protocols overwhelmingly fix big-endian order, and RFC 4506 fixed it for XDR, which is a decision in favour of decoder simplicity over encoder convenience. The argument that the choice is largely arbitrary and that agreeing matters far more than which convention is agreed was made directly by Cohen in the paper that gave the field its vocabulary.

Alignment is the same tradeoff in space. XDR pads everything to four-byte boundaries so that a reader can load fields directly, and pays in wasted bytes. For a field of size f aligned to boundary a, the padding is

$$P(f, a) = (a - (f \bmod a)) \bmod a$$

which is zero when f is already a multiple of a and approaches $a - 1$ otherwise. Across a record of many small fields the waste accumulates, and for a format carrying many one-byte flags aligned to eight bytes it can dominate the payload entirely.

Zero-copy formats must respect alignment because they hand out direct references, so they inherit the padding cost as a structural consequence rather than a choice. Formats that decode into fresh objects are free to pack tightly, because the decoder fixes up alignment as it builds.

Ossification and the Cost of Success

A format that succeeds becomes difficult to change, and the difficulty grows with the number of independent implementations that have made assumptions about it.

Ossification is the specific failure where middleboxes and intermediate implementations inspect fields they were not meant to inspect, and then break when those fields take legitimate but previously unseen values. This is not a theoretical concern, and the evidence is measured rather than argued. Honda and colleagues tested whether new TCP options could still traverse the public internet and found that a substantial fraction of paths interfered with anything unfamiliar, which is the empirical result that reframed extensibility as a deployment problem rather than a specification problem. Edeline and Donnet later quantified how path brokenness specifically affects TCP options, and Papastergiou and colleagues survey the resulting body of work and the mitigations proposed against it. RFC 9170 distils the operational lesson, which is that an extension point not exercised in practice will not work when it is finally needed, so the only reliable defence is to use the mechanism continuously rather than reserve it. The field is now effectively frozen even though the specification permits change. QUIC responded by encrypting nearly all of its transport header, as RFC 9000 describes, so that intermediaries cannot form dependencies on fields the designers intend to evolve. It went further and published the small set of properties that will not change across versions as RFC 8999, which converts an implicit and unbounded dependency surface into an explicit and bounded one. The general principle is that anything visible will eventually be depended upon, whether or not the specification permits it.

A project that treats a change to its bytecode as requiring explicit authorisation rather than as a routine version bump has recognised the same pressure early. Keleusma’s Self-Hosting Strategy records a bytecode-format change as a stop that needs a deliberate decision, which is an acknowledgement that the encoding is a contract with every artifact already compiled against it rather than an internal detail of the current compiler.

Instruction encodings ossify hardest of all, because the dependent implementations are shipped silicon that cannot be updated. Opcode space is allocated once and reclaimed almost never, and the practical consequence is that architectures reserve encoding space long before they know what will occupy it.

Version negotiation is the usual mitigation and carries its own cost. It adds a round trip or a field, it creates a downgrade surface if an attacker can influence the negotiation, and it multiplies the combinations that must be tested. With k versions supported on each side, the number of ordered pairs is k^2, while negotiation that always selects the highest common version yields only k distinct outcomes. The testing burden T therefore lies between

$$k \le T \le k^2$$

depending on how much of the pair space the negotiation can actually reach, which is why deployed systems support far fewer versions than their specifications permit.

The Schema Language and What Generates From It

A format is rarely deployed without a companion notation for describing messages, and the choice of notation constrains the format more than it first appears.

ASN.1 separated abstract syntax from encoding rules, and the modern equivalents repeat the separation. The Concise Data Definition Language, RFC 8610, describes valid CBOR without prescribing how it is written. Protocol Buffers, Thrift, Cap'n Proto, and FlatBuffers each ship an interface definition language whose primary output is generated code rather than a validation rule, which is a different emphasis with different consequences.

Generated code is fast and rigid. It turns field access into a compile-time-checked member reference, and it requires a build step, a toolchain, and a regeneration discipline whenever the schema changes. Reflective decoding is slow and flexible, since it can handle a message whose shape is discovered at run time, and it moves errors from compile time to run time. Systems that must accept messages from parties they do not control tend toward the reflective end regardless of what their performance budget would prefer.

The schema language also determines what evolution is expressible. A notation with no way to say that a field is optional cannot describe a message that adds one. A notation that cannot express a discriminated union forces the union into a convention, and conventions are not checked.

Streaming Against Whole-Message Processing

A decoder that must produce output before it has seen the whole message faces constraints that a decoder allowed to buffer does not.

Streaming forbids any construct whose interpretation depends on bytes that arrive later. A length prefix is streaming-friendly because it arrives first. A trailing checksum is not, since nothing can be trusted until it is verified, which is why protocols that stream and authenticate must either accept provisional processing or chunk the stream into independently verified units.

The constraint compounds with canonicity. A canonical format that requires map keys in sorted order cannot be produced by an encoder that discovers keys as it goes, so the encoder must buffer, and the buffering it must do is exactly what the streaming design was trying to avoid.

Where the Tradeoffs Interact

The choices are not independent, and several combinations are contradictory.

Canonical encoding conflicts with permissive handling of the unrecognised. If a reader may skip fields it does not understand, then two readers with different schemas derive different values from identical bytes, which is exactly what canonicity is meant to prevent. Consensus systems resolve this by being strict, and the strictness is a consequence of the canonicity requirement rather than an independent choice.

Zero-copy conflicts with compactness, because reading in place requires alignment and offsets that a compact encoding would remove.

Inline schemas conflict with the case for canonical encoding, since inline metadata multiplies the opportunities for two encoders to differ legitimately.

Human readability conflicts with almost everything else, which is the honest reason binary formats keep being reinvented despite the debugging cost being real.

The practical consequence is that a format cannot be evaluated against a checklist of desirable properties, because the properties are not simultaneously satisfiable. It can only be evaluated against a deployment, and the deployment determines which conflicts matter.

Epistemic State

The eleven tradeoffs are an organising choice rather than a standard list. Other treatments would divide the space differently, and the boundaries between sections are not sharp. Alignment could reasonably be folded into compactness, and ossification could be treated as a consequence of the unrecognised-handling decision rather than as a separate concern.

The equations are definitional and illustrative. They state how the schema-distribution threshold, transmission and decode time, and alignment padding are computed. They are not measurements, they use simplified models that ignore latency, caching, and instruction-level parallelism, and no benchmark in this article supports any performance claim.

The ossification section is the one place where the argument rests on measurement rather than on reasoning. The cited studies establish that unfamiliar TCP options are interfered with on a substantial fraction of internet paths, and the generalisation from that finding to wire formats as a class is this article's extrapolation rather than a claim those authors make.

The claim that the tradeoffs transfer across the three families is an argument advanced by this pair of articles, not a reported consensus. It is supported by pointing at structurally similar decisions in each family. A reader who considers the resemblance superficial rather than structural will reasonably discount the conclusion.

Statements about specific formats are drawn from their published specifications. The characterisation of ossification and the QUIC response reflects the stated design rationale, and readers should treat the general principle that visible fields attract dependencies as an empirical regularity rather than a law.

The claim that consensus systems have no choice about canonical encoding is strong and deliberate. It follows from the requirement that all participants derive identical results, and it would be falsified by a consensus design that tolerates encoding variation without divergence.

Out of Scope

Compression algorithms are excluded. Their interaction with format design is discussed only through the time model, and the algorithms themselves are a separate subject.

Cryptographic protocol design is excluded beyond the canonical-encoding requirement. Signature schemes, authenticated encryption, and downgrade-resistant negotiation each deserve separate treatment.

Parser-differential vulnerabilities are mentioned only through the unambiguity requirement. The security literature on this class is substantial and is not surveyed here.

Formal verification of encoders and decoders is excluded, as are the specification languages that support it.

Remote procedure call frameworks, database storage formats, and columnar file formats remain out of scope, as in the companion article.

No benchmarks appear, and no format is recommended over another.

Conclusion

Wire formats face a consistent set of tradeoffs regardless of whether they carry application values, conversation structure, or programs. The schema may be inline, external, or fixed, and the choice is a volume calculation. Unit boundaries may be length-prefixed, delimited, or fixed-width, and each fails in its own way. Handling of the unrecognised determines whether independent deployment is possible at all. Inspectability, canonicity, compactness, and portability each cost something specific.

The choices interact, and several combinations are contradictory. Canonicity forces strictness. Zero-copy forces padding. Success forces ossification, and the mitigations for ossification carry costs of their own.

No format is best. A format is a set of positions on these tradeoffs, and the positions are appropriate or not with respect to a deployment. The value of recognising that interchange encodings, protocol framing, and instruction encodings are one class is that a decision made carefully in one of them is a decision already understood in the others.

References

- Hennessy, John L. and Patterson, David A., Computer Architecture, A Quantitative Approach, Morgan Kaufmann
- Stevens, W. Richard, TCP/IP Illustrated, Volume 1, The Protocols, Addison-Wesley
- Tanenbaum, Andrew S. and Wetherall, David J., Computer Networks, Pearson
- Apache Avro Specification
- Cap’n Proto Encoding Specification
- ITU-T X.690, ASN.1 Encoding Rules, BER, CER and DER
- RFC 8610, Concise Data Definition Language
- RFC 8999, Version-Independent Properties of QUIC
- Protocol Buffers Field Presence Documentation
- RFC 4506, XDR, External Data Representation Standard
- RFC 7541, HPACK, Header Compression for HTTP/2
- RFC 8949, Concise Binary Object Representation
- RFC 9000, QUIC, A UDP-Based Multiplexed and Secure Transport
- RFC 9170, Long-Term Viability of Protocol Extension Mechanisms
- RFC 9413, Maintaining Robust Protocols
- RISC-V Unprivileged Specification
- Anantharaman, Prashant, Millian, Michael and Bratus, Sergey, Input Handling Done Right, IEEE SecDev, 2017
- Cohen, Danny, On Holy Wars and a Plea for Peace, Computer 14, 1981
- Edeline, Korian and Donnet, Benoit, Evaluating the Impact of Path Brokenness on TCP Options, ACM ANRW, 2020
- Honda, Michio, Nishida, Yoshifumi and Raiciu, Costin, Is It Still Possible to Extend TCP, ACM IMC, 2011
- Papastergiou, Giorgos, Fairhurst, Gorry and Ros, David, De-Ossifying the Internet Transport Layer, IEEE Communications Surveys and Tutorials 19, 2017
- Sassaman, Len, Patterson, Meredith L. and Bratus, Sergey, A Patch for Postel’s Robustness Principle, IEEE Security and Privacy 10, 2012
- Related Post, Block-Structured Control Flow and Single-Pass Validation
- Related Post, Fixup Tables and the Forward-Jump Problem
- Related Post, Getting Started with Solana Using Rust and Anchor
- Related Post, Keleusma’s Self-Hosting Strategy
- Related Post, Symbol Tables, Scope Popping, and Bounded Working Memory
- Related Post, WASM on a Jekyll Blog with Rust and wasm-bindgen
- Related Post, Wire Formats, What They Are