JSON BinPack: A space-efficient schema-driven and schema-less binary serialization specification based on JSON Schema

Juan Cruz Viotti
Kellogg College

University of Oxford
A thesis presented for the degree of
Master in Software Engineering

2022

Acknowledgements

I would like to thank my supervisor Mital Kinderkhedia for invaluable guidance and supervision for this thesis.

I would like to thank my life-partner Darlene for her immense support and patience while I worked long evenings and weekends to pursue my studies and bring this dissertation to life. Your faith in me always motivates me to be a better engineer and a better person each day.

Abstract

An increasing amount of Internet-based software systems exchange information using the JSON serialization specification through interfaces modeled with the JSON Schema definition language. By definition, this type of software systems are sensitive to substandard network performance, which translates to impaired user experience. JSON is not considered a space-efficient data interchange format. As a consequence, software systems that adopt JSON often require investing significant capital on network resources. A solution to improve network performance and reduce bills is to transmit less data through the use of a space-efficient JSON-compatible serialization specification. Following an in-depth survey into the history, characteristics and inner workings of JSON-compatible serialization specifications, I present a comprehensive space-efficiency benchmark using the SchemaStore open-source dataset of real-world JSON and JSON Schema documents across industries. Using the conclusions from the survey and benchmark studies, I propose *JSON BinPack*, a novel protocol-independent schema-driven and schema-less binary serialization specification that is strictly-compatible with JSON and takes advantage of JSON Schema formal definitions to produce bit-strings that are space-efficient in comparison to every considered alternative serialization specification.

Table of Contents

1 | Introduction

1.1 Serialization and Deserialization

Serialization is the process of translating a data structure into a *bit-string* (a sequence of bits) for storage or transmission purposes. The original data structure can be reconstructed from the bit-string using a process called *deserialization*. Serialization specifications define the bidirectional translation between data structures and bit-strings. Serialization specifications support persistence and the exchange of information in a machine-and-language-independent manner.

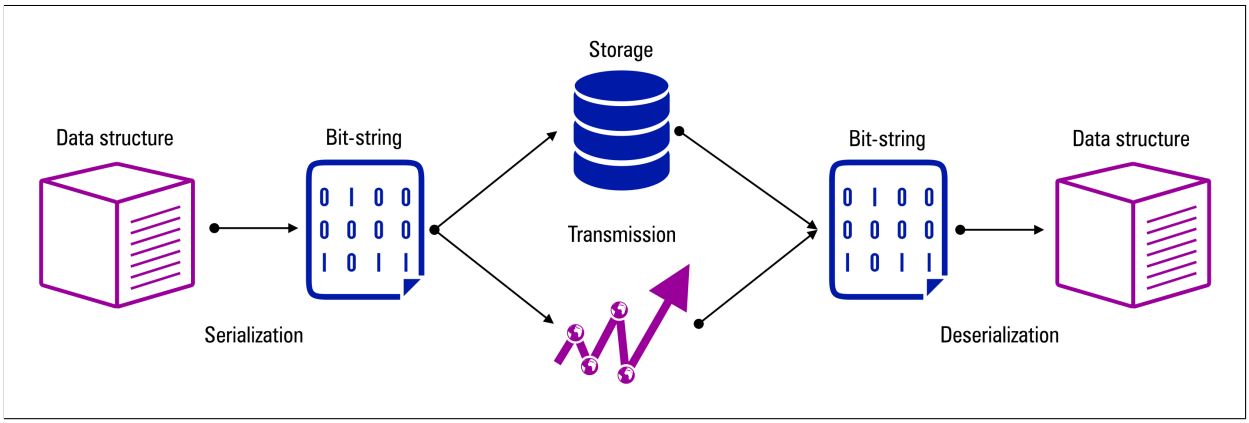

Figure 1.1: The process of translating a data structure to a bit-string is called *serialization*. The process of translating a bit-string back to its original data structure is called *deserialization*.

Serialization specifications are categorized based on how the information is represented as a bit-string i.e. *textual* or *binary* and whether the serialization and deserialization processes require a formal description of the data structure i.e. *schema-driven* or *schema-less*. Before I go into a detailed discussion about textual and binary serialization specifications, I motivate by discussing the history and evolution of serialization specifications.

1.2 History and Evolution of Serialization Specifications

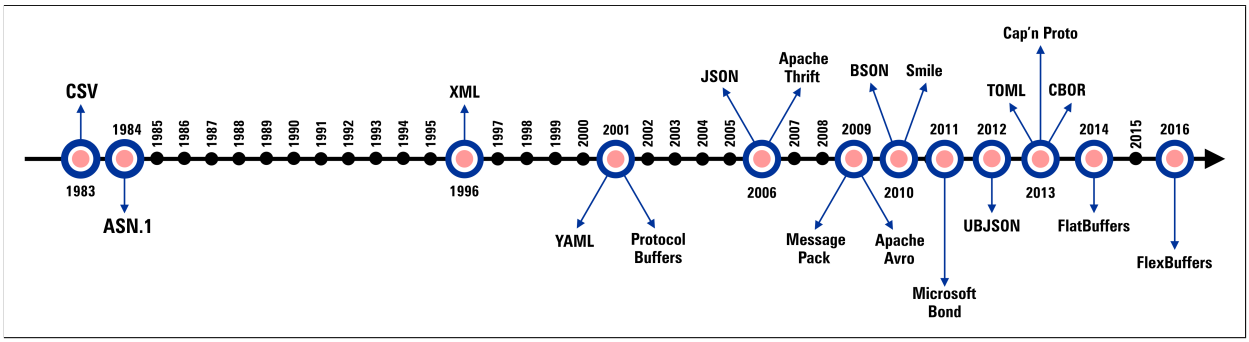

Figure 1.2: A timeline showcasing some of the most popular serialization specifications since the early 1980s.

1960s. In 1969, IBM developed *GML* (Generalized Markup Language) [1], a markup language and schema-less textual serialization specification to define meaning behind textual documents. Decades later, XML [104] was inspired by GML.

1970s. In 1972, IBM OS/360 introduced a general-purpose schema-less serialization specification as part of their FORTRAN suite [75]. The IBM FORTRAN manuals referred to the serialization specification as *List-Directed Input/Output*. It consisted of comma-separated or space-separated values that now resemble the popular CSV [127] schema-less textual serialization specification.

[1] http://www.sgmlsource.com/history/roots.htm

1980s. Microsoft invented the *INI* general purpose schema-less textual serialization specification [2] as part of their MS-DOS operating system in the early 1980s (the exact year is unclear). In 2021, the Microsoft Windows operating system continues to make use of the INI specification. INI also inspired the syntax of configuration file formats in popular software products such as *git* [3] and *PHP* [4]. In 1983, the Osborne Executive portable computer Reference Guide [39] used the term *CSV* to refer to files containing comma-separated rows of data. In 1984, the *International Telecommunication Union* [5] specified the *ASN.1* schema-driven binary serialization specification as part of the [122] standard. The ASN.1 serialization specification became a standalone standard in 1988. In 1986, the *SGML* (Standard Generalized Markup Language), a descendant of IBM GML to define custom markup languages, was proposed as an ISO standard [43]. In the late 1980s, NeXT introduced the schema-less textual ASCII-based [33] *Property List* serialization format [6] which is now popular in Apple's operating systems.

1990s. In 1996, the Netscape web browser used stringified representations of JavaScript data structures for data interchange [41]. This serialization approach would be standardized as JSON [21] a decade later. Also, the SGML [43] language inspired the first working draft of a general-purpose schema-less textual serialization specification named *XML* (Extensible Markup Language) [22]. In 1997, The Java programming language JDK 1.1 defined a *Serializable* interface [7] that provided binary, versioned and streamable serialization of Java classes and their corresponding state. This serialization specification is referred to as *Java Object Serialization* [8] and it is still in use. In 1998, [102] further improved this object serialization technique. In 1999, XML became a W3C (World Wide Web Consortium) [9] recommendation [22].

2000s. In 2000, Apple (which acquired NeXT in 1997) introduced a binary encoding for the *Property List* serialization specification [10]. A year later, Apple replaced the ASCII-based [33] original *Property List* encoding with an XML-based encoding [11]. In 2001, Google developed an internal schema-driven binary serialization specification and RPC protocol named *Protocol Buffers* [67]. In the same year, the first draft of the schema-less textual *YAML* [12] serialization specification was published as a human-friendly alternative to XML [104]. Refer to [49] for a detailed discussion of the differences between YAML and JSON. The widely-used *CSV* [127] schema-less textual serialization specification was standardized in 2005. The first draft of the *JSON* schema-less textual serialization specification was published in 2006 [40]. In the same year, Facebook developed an open-source schema-driven binary serialization specification and RPC protocol similar to Protocol Buffers [67] named *Apache Thrift* [129]. In 2008, Google open-sourced *Protocol Buffers* [67]. In 2009, the *MessagePack* [64] schema-less binary serialization specification was introduced by Sadayuki Furuhashi [12]. Two other binary serialization specifications were released in 2009: The Apache Hadoop [13] framework introduced the *Apache Avro* [61] schema-driven serialization specification. The MongoDB database [14] introduced a schema-less serialization alternative to JSON [21] named *BSON* (Binary JSON) [95].

Advances since 2010. Two new schema-less binary serialization specification alternatives to JSON [21] were conceived in 2010 and 2012: *Smile* [121] and *UBJSON* [20], respectively. In 2011, Microsoft developed *Bond* [96], a schema-driven binary serialization specification and RPC protocol inspired by Protocol Buffers [67] and Apache Thrift [129]. In 2013, the lessons learned from Protocol Buffers [67] inspired one of its original authors to create an open-source schema-driven binary serialization specification and RPC protocol named *Cap'n Proto* [146]. Two schema-less serialization specifications were created on 2013: a textual serialization specification inspired by INI named *TOML* [114] and a binary serialization specification designed for the Internet of Things named *CBOR* [17]. In 2014, Google released *FlatBuffers* [143], a schema-driven binary serialization specification that was later found to share some similarities to Cap'n Proto [146]. In 2015, Microsoft open-sourced *Bond* [96]. In 2016, Google introduced

[2] https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc731332(v=ws.11)?redirectedfrom=MSDN
[3] https://git-scm.com
[4] https://www.php.net
[5] https://www.itu.int/ITU-T/recommendations/index.aspx?ser=X
[6] https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/PropertyLists/OldStylePlists/OldStylePLists.html
[7] https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html
[8] https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
[9] https://www.w3.org
[10] https://opensource.apple.com/source/CF/CF-1153.18/CFBinaryPList.c.auto.html
[11] https://web.archive.org/web/20140219093104/http://www.appleexaminer.com/MacsAndOS/Analysis/PLIST/PLIST.html
[12] https://github.com/frsyuki
[13] https://hadoop.apache.org
[14] https://www.mongodb.com

FlexBuffers [144], a schema-less variant of FlatBuffers [143].

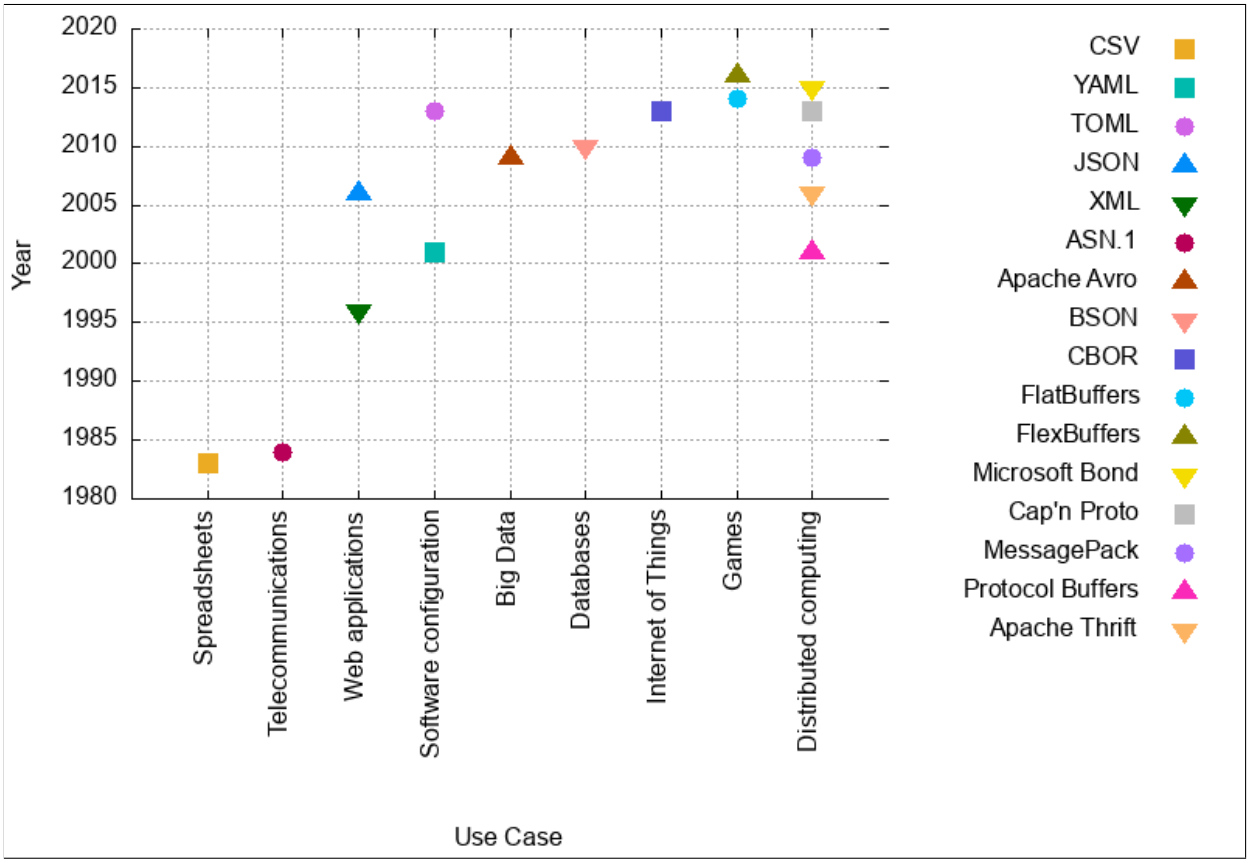

Figure 1.3: The most popular serialization specifications by their use case.

[15] https://www.oss.com/company/customers.html
[16] https://lists.apache.org/thread.html/rc11fcfbc294bb064c6e59167f21b38f3eb6d14e09b9af60970b237d6%40%3Cuser.avro.apache.org%3E
[17] https://www.linkedin.com/in/kenton-varda-5b96a2a4/
[18] https://github.com/google/flatbuffers/issues/6424
[19] https://github.com/google/flatbuffers/network/dependents
[20] https://groups.google.com/g/protobuf/c/tJVbWK3y_TA/m/vpOiSFfqAQAJ
[21] https://github.com/protocolbuffers/protobuf/network/dependents
[22] https://thrift.apache.org/about#powered-by-apache-thrift
[23] https://www.npmjs.com/package/tinycbor
[24] https://github.com/OneNoteDev/GoldFish
[25] https://github.com/outfoxx/PotentCodables
[26] https://github.com/PelionIoT/cbor-sync
[27] https://github.com/msgpack/msgpack-node/network/dependents
[28] https://github.com/msgpack/msgpack-ruby/network/dependents
[29] https://github.com/msgpack/msgpack-javascript/network/dependents
[30] https://github.com/msgpack/msgpack/issues/295
[31] https://github.com/FasterXML/smile-format-specification/issues/15
[32] https://docs.teradata.com/reader/C8cVEJ54PO4~YXWXeXGvsA/b9kd0QOTMB3uZp9z5QT2aw
[33] https://reference.wolfram.com/language/ref/format/UBJSON.html

Table 1.1: A non-exhaustive list of companies that publicly state that they are using the binary serialization specifications discussed in this paper.

Serialization Specification	**Companies**
ASN.1	Broadcom, Cisco, Ericsson, Hewlett-Packard, Huawei, IBM, LG Electronics, Mitsubishi, Motorola, NASA, Panasonic, Samsung, Siemens [15]
Apache Avro	Bol, Cloudera, Confluent, Florida Blue, Imply, LinkedIn, Nuxeo, Spotify, Optus, Twitter [16]
Microsoft Bond	Microsoft
Cap'n Proto	Sandstorm, Cloudflare [17]
FlatBuffers / FlexBuffers	Apple, Electronic Arts, Facebook, Google, Grafana, JetBrains, Netflix, Nintendo, NPM, NVidia, Tesla [18], [19]
Protocol Buffers	Alibaba, Amazon, Baidu, Bloomberg, Cisco, Confluent, Datadog, Dropbox, EACOMM, Facebook, Google, Huawei, Intel, Lyft, Microsoft, Mozilla, Netflix, NVidia, PayPal, Sony, Spotify, Twitch, Uber, Unity, Yahoo, Yandex [20], [21]
Apache Thrift	Facebook, Cloudera, Evernote, Mendeley, last.fm, OpenX, Pinterest, Quora, RapLeaf, reCaptha, Siemens, Uber [22]
BSON	MongoDB
CBOR	Intel [23], Microsoft [24], Outfox [25], Pelion [26]
MessagePack	Amazon, Atlassian, CODESYS, Datadog, Deliveroo, GitHub, Google, GoSquared, LinkedIn, Microsoft, Mozilla, NASA, National Public Radio, NPM, Pinterest, Sentry, Shopify, Treasure Data, Yelp [27], [28], [29], [30]
Smile	Ning, Elastic [31]
UBJSON	Teradata [32], Wolfram [33]

1.3 Textual and Binary Serialization Specifications

A serialization specification is *textual* if the bit-strings it produces correspond to sequences of characters in a text encoding such as ASCII [33], EBCDIC/CCSID 037 [76], or UTF-8 [38], otherwise the serialization specification is *binary*.

We can think of a textual serialization specification as a set of conventions within the boundaries of a text encoding such as UTF-8 [38]. The availability and diversity of computer tools to operate on popular text encodings makes textual serialization specifications to be perceived as human-friendly. In comparison, binary serialization specifications are not constrained by a text encoding. This flexibility typically results in efficient representation of data at the expense of requiring accompanying documentation and specialized tools.

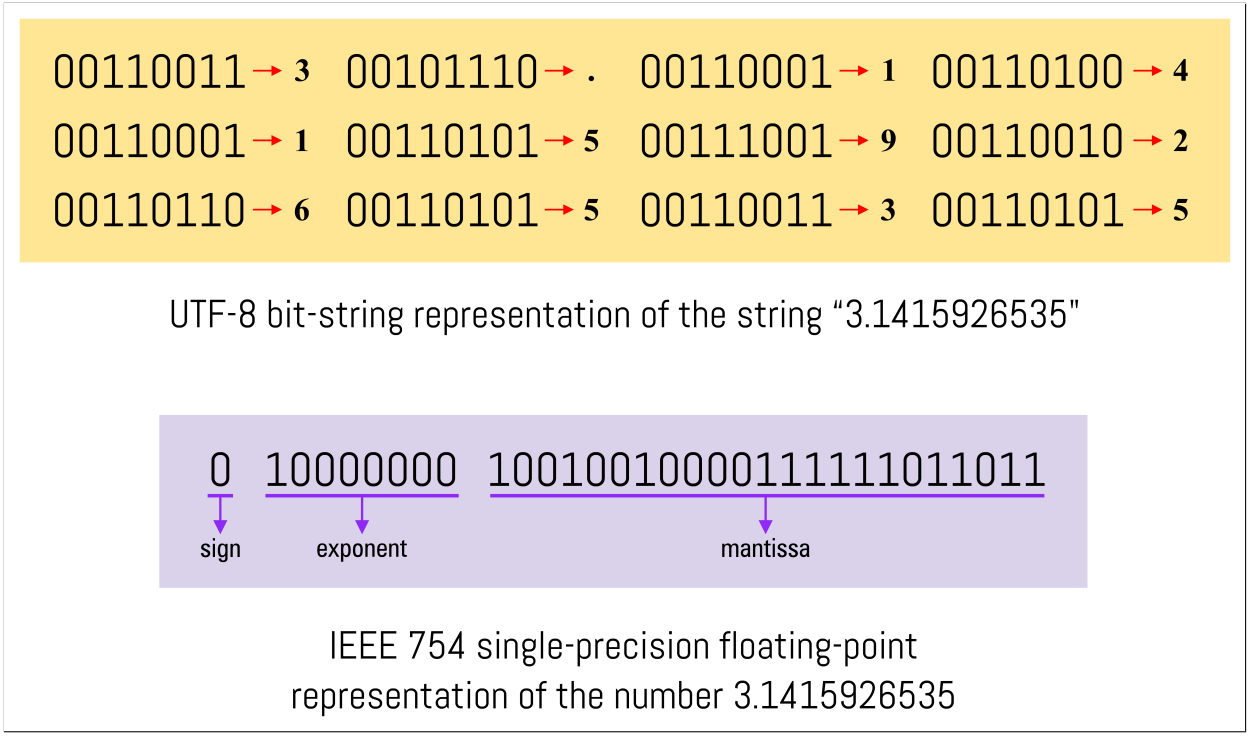

Figure 1.4: Textual and binary representations of the decimal number 3.1415926535. The textual representation encodes the decimal number as a 96-bits sequence of numeric characters ("3" followed by ".", followed by "1", and so forth) that we can inspect and understand using a text editor. The binary representation encodes the decimal number in terms of its sign, exponent, and mantissa. The resulting bit-string is only 32 bits long - three times smaller than the textual representation. However, we are unable to understand it using generally-available text-based tools.

1.4 Schema-less and Schema-driven Serialization Specifications

A *schema* is a formal definition of a data structure. For example, a schema may describe a data structure as consisting of two Big Endian IEEE 754 single-precision floating-point numbers [60]. A serialization specification is *schema-less* if it produces bit-strings that can be deserialized without prior knowledge of its structure and *schema-driven* otherwise.

Implementations of schema-less serialization specifications embed the information provided by a schema into the resulting bit-strings to produce bit-strings that are standalone with respect to deserialization. In comparison to schema-driven serialization specifications, schema-less serialization specifications are perceived as easier to use because receivers can deserialize any bit-string produced by the implementation and not only the ones the receivers know about in advance. Alternatively, schema-driven specification implementations can avoid encoding certain structural information into the bit-strings they produce. This typically results in compact space-efficient bit-strings. For

this reason, network-efficient systems tend to adopt schema-driven serialization specifications [112]. Schema-driven serialization specifications are typically concerned with space efficiency and therefore tend to be binary. However, [31] propose a textual JSON-compatible schema-driven serialization specification. In the case of communication links with large bandwidths or small datasets, the gains are negligible but considering slow communication links or large datasets which could be terabytes in size, the choice of serialization specification could have a big impact.

Writing and maintaining schema definitions is a core part of using schema-driven serialization specifications. Most schema-driven serialization specifications implement a custom schema language that is not usable by any other schema-driven serialization specification. A schema-driven serialization specification may use a low-level or a high-level schema definition language. Low-level schema definition languages such as *PADS* [58], *BinX* [154], and *Data Format Description Language* (DFDL) [154] are concerned with describing the contents of bit-strings while high-level schema definition languages such as *ASN.1* [123] and *JSON Schema* [158] abstractly describe data structures and depend on the serialization specification implementation to provide the corresponding encoding rules.

Often, schemas are auto-generated from formal definitions such as database tables [34], other schema languages [35], or inferred from the data [62] [84] [6] [42] [157] [48] [28] [135] [5]. There are also examples of domain-specific schema-driven serialization specifications where the schema definition is implicitly embedded into the serialization and deserialization implementation routines, such as the SOL binary representation for sensor measurements [24].

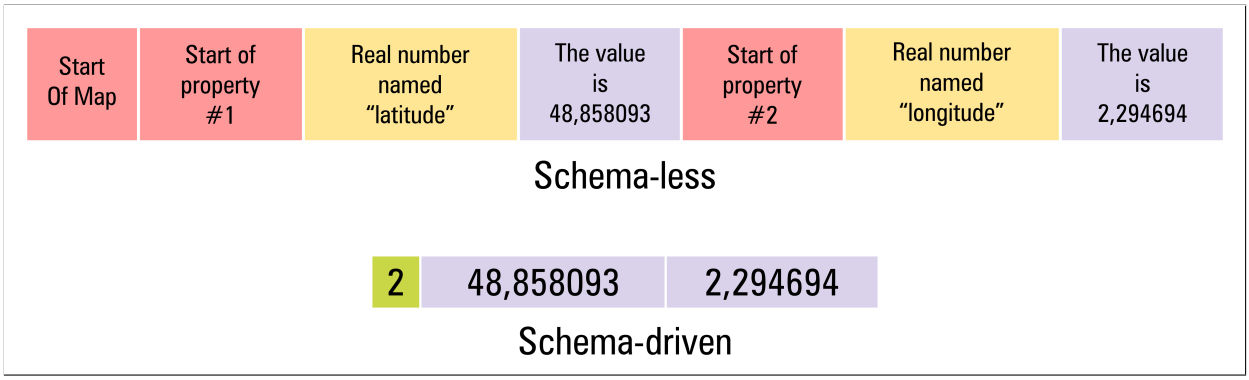

Figure 1.5: An associative array (also known as a map) that consists of two decimal number properties, "latitude" and "longitude", serialized with fictitious schema-less and schema-driven serialization specifications. The schema-less representation (top) is self-descriptive and each property is self-delimited. In comparison, the schema-driven representation (bottom) omits most self-descriptive information except for the length of the associative array as an integer prefix. A reader cannot understand how the schema-driven representation translates to the original data structure without additional information such as a schema definition.

1.5 Schema-less as a Subset of Schema-driven

Schema-driven serialization specifications avoid embedding structural information into the resulting bit-strings for space-efficiency purposes. If a schema fails to capture essential structural information then the serialization specification has to embed that information into the resulting bit-strings. We can reason about schema-less serialization specifications as schema-driven specifications where the schema is generic enough that it describes any bit-string and as a consequence carries no substantial information about any particular instance. For example, a schema that defines bit-strings as sequences of bits can describe any bit-string while providing no useful information for understanding the semantics of such bit-strings.

The amount of information included in a schema can be thought as being *inversely proportional* to the information that needs to be encoded in a bit-string described by such schema. However, schema-driven serialization specifications may still embed redundant information into the bit-strings with respect to the schema for runtime-efficiency, compatibility or error tolerance. We can rank schema-driven serialization specifications based on the extent of information that is necessary to include in the bit-strings.

[34] https://github.com/SpringTree/pg-tables-to-jsonschema
[35] https://github.com/okdistribute/jsonschema-protobuf

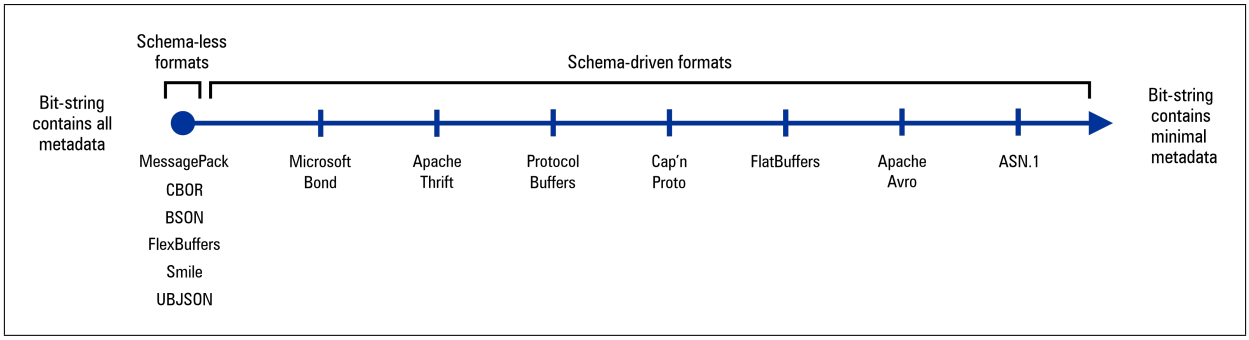

Figure 1.6: We can compare schema-less and schema-driven serialization specifications based on how much information their resulting bit-strings contain. Schema-less specifications are equivalent in that they all consist of implicit schemas that convey no information. However, not all schema languages supported by schema-driven specifications can describe the same amount of information. For this reason, some schema-driven specifications need to encode more structural information into their resulting bit-strings than others, placing them on the left hand side of the line. Schema-driven specifications that enable meticulously defined schemas are placed on the right hand side of the line.

1.6 The Relevance of Space-efficiency

An increasing amount of software is now accessed over the Internet. As a consequence, these software systems are particularly sensitive to substandard network performance. Given the decentralized architecture and complex dynamics of the Internet, network communication is unpredictable and often unreliable. For consumers of Internet-based software systems, substandard network performance results in impaired user experience. For example, Google found that "*half a second (page load) delay caused 20% drop in traffic*" [36] as early as in 2006. More recently, in 2017, Akamai, a global content delivery network (CDN), found that "*a 100-millisecond delay in website load time can hurt conversion rates by 7 percent*" and that "*a two-second delay in web page load time increases bounce rates by 103 percent*" [37]. Additionally, software systems that operate over the Internet typically rely on *infrastructure as a service* (IaaS) or *platform as a service* (PaaS) providers such as Amazon Web Services (AWS) [38] and Microsoft Azure [39]. These providers typically operate on a pay-as-you-go model where they charge per resource utilization. Therefore, transmitting data over the network directly translates to operational expenses.

For interoperability purposes, Internet-based software systems transmit information using data serialization specifications such as JSON [46] and XML [104]. [151] identifies JSON [46] as the dominant data interchange standard in the context of cloud software systems. However, it concludes that JSON is not a space-efficient serialization specification. [86] argues that network communication is time-expensive and that the network communication bottleneck *makes computation essentially free in comparison*. In the context of HTTP/1.1 [55], it argues that the computational overhead of making payloads space-efficient using techniques such as data compression are minimal compared to the time-overhead of a low bandwidth network connection, which are still common according to [120]. Therefore, in comparison to JSON [46], the use of space-efficient serialization specifications to transmit less data reduces operational costs of Internet-based software systems and is a key factor to improve network performance and user experience.

1.7 Contributions

For this dissertation, a research-oriented approach was pursued. The contributions are as follows:

- **Paper - A Survey of JSON-compatible Binary Serialization Specifications [151]**, joint-paper with supervisor, Mital Kinderkhedia. Submitted for publication. This paper consists of an in-depth study of the history, characteristics, advantages and capabilities of the space-efficient encodings of 13 popular JSON-compatible

[36] http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html
[37] https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp
[38] https://aws.amazon.com
[39] https://azure.microsoft.com

binary serialization specifications: ASN.1 [123], Apache Avro [61], Microsoft Bond [96], Cap'n Proto [146], FlatBuffers [143], Protocol Buffers [67], Apache Thrift [129], BSON [95], CBOR [17], FlexBuffers [144], MessagePack [64], Smile [121] and UBJSON [20]. I believe this work is the first of its kind to introduce annotated hexadecimal bit-strings that help the reader understand the inner workings and optimizations that these binary serialization specifications perform when serializating a non-trivial JSON [46] document.

- **Open Source Contributions.** Through the process of conducting a literature review, I identified and resolved 13 issues with the documentation and specifications of the Apache Avro [61], Apache Thrift [129], FlatBuffers [143], FlexBuffers [144], Microsoft Bond [96], and Smile [121] binary serialization open-source projects. The fixes, corresponding patches and links are listed in Table 1.2 and Table 1.3.

- **Schema Evolution Compatibility Analysis.** I present and discuss an in-depth study of the schema evolution characteristics of 7 popular JSON-compatible schema-driven serialization specifications: ASN.1 [123], Apache Avro [61], Microsoft Bond [96], Cap'n Proto [146], FlatBuffers [143], Protocol Buffers [67] and Apache Thrift [129]. The study (see Appendix C) consists of testing 35 type conversion, enumerations, unions, lists and structural schema transformations that either confine, expand, change or preserve the domain of the schemas. For every case, I document whether the given schema-driven binary serialization specification fails under such transformation or supports it as a fully-compatible, backwards-compatible or forwards-compatible schema transformation.

- **Paper - A Benchmark of JSON-compatible Binary Serialization Specifications [150]**, joint-paper with supervisor, Mital Kinderkhedia. Submitted for publication. This paper presents and discusses a space-efficiency benchmark of the 13 JSON-compatible binary serialization specifications studied in [151] using a dataset of 27 JSON [46] documents methodically selected using the taxonomy for JSON documents introduced in chapter 4. I believe this work is the first of its kind to produce a comprehensive, reproducible, extensible and open-source space-efficiency benchmark of JSON-compatible binary serialization specifications that considers a representative input dataset of real-world JSON [46] documents across industries.

- **Taxonomy for JSON Documents.** Through the process of conducting a literature review of space-efficiency benchmarks involving JSON-compatible serialization specifications, I identified a lack of a methodical approach for selecting representative sets of input JSON [46] documents for benchmarking purposes. I believe this work is the first of its kind to introduce a formal tiered taxonomy (see chapter 4) consisting of 36 categories as a common basis to class JSON [46] documents based on their size, type of content, characteristics of their structure and redundancy criteria.

- **JSON Stats Online Tool.** I developed a publicly-available web application called *JSON Stats* to automatically categorize JSON [46] documents according to the taxonomy introduced in chapter 4. This online tool is available at `https://www.jsonbinpack.org/stats/` and its source code is publicly-available on GitHub [40] under the Apache-2.0 software license [41]. Refer to Figure 4.10 for a screenshot of the tool in action.

- **Extensible Space-efficiency Benchmark Software.** Through the process of conducting a literature review, I identified a lack of an industry-standard automated software to define JSON-compatible space-efficiency benchmarks. As a solution, I designed and implemented an extensible, automated and deterministic benchmark platform to declare JSON [46] input documents, declare JSON-compatible serialization specifications written in arbitrary programming languages, declare data compression formats, extract raw and aggregate statistical data and generate plots to visualize the results. The benchmark software is publicly-available on GitHub [42] under the Apache-2.0 software license. The benchmark runs on the cloud using GitHub Actions [43] and the results are automatically published to `https://www.jviotti.com/binary-json-size-benchmark/`. This benchmark platform is used in chapter 7 to evaluate the 13 JSON-compatible serialization specifications listed in chapter 6 and further extended to evaluate JSON BinPack in chapter 9.

- **JSON BinPack.** The core contribution of this thesis is the design and development of a novel and space-efficient strictly-JSON-compatible binary serialization specification based on JSON Schema [159] called *JSON*

[40] `https://github.com/jviotti/jsonbinpack`
[41] `https://www.apache.org/licenses/LICENSE-2.0.html`
[42] `https://github.com/jviotti/binary-json-size-benchmark`
[43] `https://github.com/features/actions`

BinPack (see chapter 8). The JSON BinPack specification and its first proof-of-concept implementation are available for free on GitHub [44] under the Apache-2.0 open-source software license. To the best of my knowledge, JSON BinPack is the first schema-driven binary serialization specification to adopt JSON Schema [159] as its schema definition language.

Table 1.2: A list of open-source contributions made by the author in the process of writing this survey paper. This table is continued in Table 1.3.

Specification	Repository	Commit	Description
Apache Avro [61]	https://github.com/apache/avro	afe8fa1	Improve the encoding specification to clarify that records encode fields even if they equal their explicitly-set defaults and that the `default` keyword is only used for schema evolution purposes
Apache Thrift [129]	https://github.com/apache/thrift	2e7f39f	Improve the Compact Protocol specification to clarify the Little Endian Base 128 (LEB128) variable-length integer encoding procedure and include a serialization example
Apache Thrift [129]	https://github.com/apache/thrift	47b3d3b	Improve the Compact Protocol specification to clarify that strings are not delimited with the *NUL* ASCII [33] character
FlatBuffers [143]	https://github.com/google/flatbuffers	4aff119	Extend the binary encoding specification to document how union types are encoded
FlatBuffers [143]	https://github.com/google/flatbuffers	7b1ee31	Improve the documentation to clarify that the schema language does not permit unions of scalar types but that the C++ [79] implementation has experimental support for unions of structs and strings
FlatBuffers [143]	https://github.com/google/flatbuffers	52e2177	Remove from the documentation an outdated claim that Protocol Buffers [67] does not support union types
FlatBuffers [143]	https://github.com/google/flatbuffers	796ed68	Improve the FlatBuffers [143] and FlexBuffers [144] encoding specifications to clarify that neither specifications deduplicate vector elements but that vectors may include more than one offset pointing to the same value

1.8 Thesis Organization

This thesis is organized as follows. In chapter 1, I present the concept of data serialization, the history and evolution of serialization specifications and the categorization of serialization specifications into textual or binary and schema-less or schema-driven. In chapter 2, I discuss, summarize and present existing literature covering JSON-compatible binary serialization specifications. In chapter 3, I present the history, characteristics, shortcomings and relevance of the JSON textual schema-less serialization specification and the JSON Schema definition language. In chapter 4, I introduce the SchemaStore real-world open-source JSON dataset and define a taxonomy for JSON documents consisting of 36 tierer categories as a common basis to classify JSON documents based on their size, type of content, redundancy criteria and structural characteristics. In chapter 5, I discuss a methodical approach to study the characteristics, optimizations and encodings of existing JSON-compatible binary serialization specifications, measure existing JSON-compatible binary serialization specifications for space-efficiency and apply those findings to introduce a new space-efficient JSON-compatible binary serialization specification. In chapter 6, I apply the methodology approach discussed in chapter 5, present an analysis example of 1 out of 13 JSON-compatible binary serialization specifications, refer

[44] https://github.com/jviotti/jsonbinpack

Table 1.3: Continuation of Table 1.2.

Specification	Repository	Commit	Description
Microsoft Bond [96]	https://github.com/microsoft/bond	4acf83b	Improve the documentation to explain how to enable the Compact Binary version 2 encoding in the C++ [79] implementation
Microsoft Bond [96]	https://github.com/microsoft/bond	0012d99	Improve the Compact Binary encoding specification to clarify that ID bits are encoded as Big Endian unsigned integers, that signed 8-bit integers use Two's Complement [59], formalize the concept of variable-length integers as Little Endian Base 128 (LEB128) encoding, clarify that real numbers are encoded as IEEE 764 floating-point numbers [60], and that enumeration constants are encoded as signed 32-bit integers
Smile [121]	https://github.com/FasterXML/smile-format-specification	ac82c6b	Fix the fact that the specification refers to ASCII [33] strings of 33 to 64 bytes and Unicode [38] strings of 34 to 65 bytes using two different naming conventions
Smile [121]	https://github.com/FasterXML/smile-format-specification	7a53b0a	Improve the specification by adding an example of how real numbers are represented using the custom 7-bit variant of IEEE 764 floating-point number encoding [60]
Smile [121]	https://github.com/FasterXML/smile-format-specification	95133dd	Improve the specification to clarify how the byte-length-prefixes of the *Tiny Unicode* and *Small Unicode* string encodings are computed differently compared to their ASCII [33] counterparts
Smile [121]	https://github.com/FasterXML/smile-format-specification	c56793f	Clarify that the encoding attempts to reserve the **`0xff`** byte for message framing purposes but that reserving such byte is no longer a requirement to make the format suitable for use with WebSockets

the reader to my paper *A Survey of JSON-compatible Binary Serialization Specifications* [151] for the remaining 12 analysis and discuss the conclusions derived from this study. In chapter 7, I apply the methodology approach discussed in chapter 5, present benchmark results for 3 input documents out of a dataset of 27 real-world JSON documents serialized using the 13 JSON-compatible binary serialization specifications studied in chapter 6, refer the reader to my paper *A Benchmark of JSON-compatible Binary Serialization Specifications* [150] for the remaining 24 benchmark results and discuss the conclusions derived from this benchmark. In chapter 8, I present and discuss the requirements specification, software design, architecture diagrams and a proof-of-concept implementation of JSON BinPack, a novel and space-efficient JSON-compatible binary serialization specification based on JSON Schema. In chapter 9, I extend the benchmark presented in chapter 7 to compare JSON BinPack against the 13 JSON-compatible binary serialization specifications studied in chapter 6 and discuss the benchmark results. In chapter 10, I present the problem of schema evolution in the context of data serialization, study the approaches to schema evolution adopted by schema-driven binary serialization specifications and present an approach to address the schema evolution problem in JSON BinPack using JSON Schema. In chapter 11, I present reflections that encompass both successes and challenges in designing and developing the JSON BinPack binary serialization specification. In chapter 12, I present my views on future directions for work on the JSON BinPack binary serialization specification and the problem of JSON-compatible space-efficient serialization.

2 | Related Literature

2.1 JSON-compatible Binary Serialization Specifications

Table 2.1 and Table 2.2 list existing literature that discusses different sets of serialization specifications, both textual and binary, schema-less and schema-driven. However, many of these publications discuss serialization specifications that are either not JSON-compatible, cannot be considered general-purpose serialization specifications, or are out of date. For example, Java Object Serialization, as its name implies, is only concerned with serialization of object instances in the Java programming language. The first Protocol Buffers [67] version 3 release was published on GitHub in 2014 [1], yet there are several publications listed in Table 2.1 and Table 2.2 released before that year discussing the now-obsolete Protocol Buffers version 2 [155] [66] [92] [133] [145] [53]. As another example, [97] discusses an ASN.1 [123] encoding (LWER) that has been abandoned in the 1990s [89].

Table 2.1: A list of publications that discuss binary serialization specifications. This table is continued in Table 2.2.

Publication	Year	Context	Discussed Serialization Specifications
An overview of ASN.1 [99]	1992	Networking	ASN.1 BER [125], ASN.1 PER [124]
Efficient encoding rules for ASN.1-based protocols [97]	1994	Networking	ASN.1 BER [125], ASN.1 CER [125], ASN.1 DER [125], ASN.1 LWER [89], ASN.1 PER [124]
Evaluation of Protocol Buffers as Data Serialization Format for Microblogging Communication [155]	2011	Microblogging	JSON [46], Protocol Buffers [67]
Impacts of Data Interchange Formats on Energy Consumption and Performance in Smartphones [66]	2011	Mobile	JSON [46], Protocol Buffers [67], XML [104]
Performance evaluation of object serialization libraries in XML, JSON and binary formats [92]	2012	Java Object Serialization	Apache Avro [61], Apache Thrift [129], Java Object Serialization [2], JSON [46], Protocol Buffers [67], XML [104]
A comparison of data serialization formats for optimal efficiency on a mobile platform [133]	2012	Mobile	Apache Thrift [129], JSON [46], Protocol Buffers [67], XML [104]
Performance evaluation of Java, JavaScript and PHP serialization libraries for XML, JSON and binary formats [145]	2012	Web Services	Apache Avro [61], Java Object Serialization [3], JSON [46], MessagePack [64], Protocol Buffers [67], XML [104]
Google protocol buffers research and application in online game [53]	2013	Gaming	Protocol Buffers [67]
Integrating a System for Symbol Programming of Real Processes with a Cloud Service [87]	2015	Web Services	JSON [46], MessagePack [64], XML [104], YAML [12]

[1] https://github.com/protocolbuffers/protobuf/releases/tag/v3.0.0-alpha-1
[2] https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
[3] https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
[4] https://www.boost.org/doc/libs/1_55_0/libs/serialization/doc/index.html
[5] https://github.com/RuedigerMoeller/fast-serialization
[6] http://hessian.caucho.com/doc/hessian-serialization.html
[7] https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
[8] https://github.com/EsotericSoftware/kryo
[9] https://github.com/protostuff/protostuff
[10] https://www.rdfhdt.org
[11] https://github.com/niXman/yas

Table 2.2: Continuation of Table 2.1.

Publication	Year	Context	Discussed Serialization Specifications
Serialization and deserialization of complex data structures, and applications in high performance computing [162]	2016	Scientific Computing	Apache Avro [61], Boost::serialize [4], Cap'n Proto [146], OpenFPM Packer/Unpacker [77], Protocol Buffers [67]
Smart grid serialization comparison: Comparison of serialization for distributed control in the context of the internet of things [109]	2017	Internet of Things	Apache Avro [61], BSON [95], CBOR [17], FST [5], Hessian [6], Java Object Serialization [7], Kryo [8], MessagePack [64], Protocol Buffers [67], ProtoStuff [9], Smile [121], XML [104], YAML [12]
Binary Representation of Device Descriptions: CBOR versus RDF HDT [119]	2018	Internet of Things	CBOR [17], JSON [46], RFD HDT [10]
Evaluating serialization for a publish-subscribe based middleware for MPSoCs [70]	2018	Embedded Development	FlatBuffers [143], MessagePack [64], Protocol Buffers [67], YAS [11]
Analytical Assessment of Binary Data Serialization Techniques in IoT Context [14]	2019	Internet of Things	BSON [95], FlatBuffers [143], MessagePack [64], Protocol Buffers [67]
FlatBuffers Implementation on MQTT Publish/Subscribe Communication as Data Delivery Format [113]	2019	Internet of Things	CSV [127], FlatBuffers [143], JSON [46], XML [104]
Enabling Model-Driven Software Development Tools for the Internet of Things [82]	2019	Internet of Things	Apache Avro [61], Apache Thrift [129], FlatBuffers [143], JSON [46], Protocol Buffers [67]
Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV [115]	2020	Automobile	Cap'n Proto [146], FlatBuffers [143], Protocol Buffers [67]
Performance Analysis and Optimization of Serialization Techniques for Deep Neural Networks [105]	2020	Machine Learning	FlatBuffers [143], Protocol Buffers [67]

2.2 Space-efficiency Benchmarks

The existing space-efficiency benchmarks involving the binary serialization specifications discussed in [151] are summarized in Table 2.3. As demonstrated in Table 2.3, schema-less binary data serialization specifications such as BSON [95], CBOR [17], MessagePack [64], and Smile [121] tend to produce smaller bit-strings than JSON (up to 63% size reduction compared to JSON). However, there are some exceptions. For instance, [112] and [109] found that BSON [95] and CBOR [17] tend to produce larger bit-strings than JSON [46] for a subset of their input data (up to 32% larger than JSON).

In comparison to schema-less serialization specifications, as demonstrated in Table 2.3, schema-driven serialization specifications such as Protocol Buffers [67], Apache Thrift [129], Apache Avro [61] tend to produce bit-strings that are up to 95% smaller than JSON. However, in this case, there are also exceptions. For instance, [109] and [145] found the MessagePack [64] schema-less binary serialization specification to be space-efficient in comparison to Protocol Buffers [67] and Apache Avro [61] (up to 23% size reduction). Similarly, [70] and [14] found MessagePack [64] to be space-efficient in comparison to Protocol Buffers [67] and FlatBuffers [143] for certain cases.

Data compression is another approach to achieve space efficiency. [66] conclude that compressed JSON is space-efficient in comparison to both compressed and uncompressed Protocol Buffers. [109] conclude that compressed JSON is space-efficient in comparison to Protocol Buffers, Smile [121], compressed and uncompressed CBOR [17], and compressed and uncompressed BSON [95]. However, Apache Avro [61], MessagePack [64], compressed Protocol Buffers, and compressed Smile are space-efficient in comparison to compressed JSON.

2.2.1 Shortcomings

I found several aspects of the existing literature to be insufficient, leading to gaps in the JSON-compatible binary serialization space-efficiency benchmark literature for the following reasons:

Coverage of Serialization Specifications. The binary serialization specifications covered by existing benchmarks are Protocol Buffers [67], MessagePack [64], FlatBuffers [143], and to a lesser extent: Apache Avro [61], Apache Thrift [129], BSON [95], CBOR [17] and Smile [121]. My previous work [151] also discusses ASN.1 [123], Microsoft Bond [96], Cap'n Proto [146], FlexBuffers [144], and UBJSON [20]. To the best of my knowledge, these have not been considered by existing space-efficiency benchmark literature.

Reproducibility and Representativity. Neither [66], [112], [109], [119], [82], nor [115] disclose the JSON [46] documents or schema definitions used to arrive at the result of their space-efficiency benchmarks. Therefore, it is not possible to corroborate their findings or contextualize their results. The publications that disclose the input JSON [46] documents are [92] and [87]. However, they are limited in scope as they consider a single JSON document in each of their papers. Other publications disclose schema definitions of varying formality that describe the JSON documents used as part of their benchmarks. Of those, [155] and [113] are concerned with one type of JSON document, [133] and [53] are concerned with two types of JSON documents, and [70] and [14] are concerned with three types of JSON documents. Therefore, I consider the results from these publications to be either not reproducible or not representative of the variety of JSON documents that are widely used in practice across different industries.

Data Compression. I found two publications that take data compression into account: [66] and [109]. However, they are limited in scope as these papers discuss only the GZIP [44] data compression format and there is no mention of the implementation used and the compression level that GZIP is configured with.

Out-of-date. Some of the existing benchmarks measure obsolete versions of certain binary serialization specifications. For example, the Protocol Buffers [67] version 3 was first released in 2014 [12]. However, there are a number of benchmark publications released before that year that discuss the now-obsolete Protocol Buffers version 2 [66] [155] [92] [133] [53].

[12] https://github.com/protocolbuffers/protobuf/releases/tag/v3.0.0-alpha-1

Table 2.3: A list of space-efficiency benchmark publications that involve JSON [46] and/or a subset of the binary serialization specifications dicussed in [151]. The third column summarises the benchmark conclusions. In this table, a serialization specification is *greater than* another serialization specification if it produced larger bit-strings in the respective publication findings.

Publication	**Year**	**Conclusion**
Impacts of data interchange formats on energy consumption and performance in smartphones [66]	2011	JSON > Protocol Buffers > Protocol Buffers with GZIP > JSON with GZIP
Evaluation of Protocol Buffers as Data Serialization Format for Microblogging Communication [155]	2011	JSON > Protocol Buffers
Performance evaluation of object serialization libraries in XML, JSON and binary formats [92]	2012	JSON > Apache Thrift > Protocol Buffers > Apache Avro
A comparison of data serialization formats for optimal efficiency on a mobile platform [133]	2012	JSON > Apache Thrift > Protocol Buffers
Google protocol buffers research and application in online game [53]	2013	JSON > Protocol Buffers
Integrating a system for symbol programming of real processes with a cloud service [87]	2015	JSON > MessagePack
Performance evaluation of using Protocol Buffers in the Internet of Things communication [112]	2016	**In most cases**: JSON > BSON > Protocol Buffers. **However, in some cases**: BSON > JSON > Protocol Buffers
Smart grid serialization comparison: Comparison of serialization for distributed control in the context of the Internet of Things [109]	2017	BSON > CBOR > JSON > BSON with GZIP > Smile > Protocol Buffers > CBOR with GZIP > JSON with GZIP > Apache Avro > Protocol Buffers with GZIP > Smile with GZIP > MessagePack > Apache Avro with GZIP > MessagePack with GZIP
Binary Representation of Device Descriptions: CBOR versus RDF HDT [119]	2018	JSON > CBOR
Evaluating Serialization for a Publish-Subscribe Based Middleware for MPSoCs [70]	2018	FlatBuffers > Protocol Buffers > MessagePack
Performance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats [145]	2018	JSON > MessagePack > Protocol Buffers > Apache Avro
Analytical assessment of binary data serialization techniques in IoT context (evaluating protocol buffers, flat buffers, message pack, and BSON for sensor nodes) [14]	2019	**For numeric and mixed data**: BSON > FlatBuffers > MessagePack > Protocol Buffers. **For textual data**: FlatBuffers > BSON > MessagePack > Protocol Buffers
Enabling Model-Driven Software Development Tools for the Internet of Things [82]	2019	JSON > FlatBuffers
Flatbuffers Implementation on MQTT Publish/Subscribe Communication as Data Delivery Format [113]	2019	JSON > FlatBuffers
Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV [115]	2020	JSON > FlatBuffers > Protocol Buffers

3 | Background

3.1 JSON

3.1.1 History and Characteristics

JSON (JavaScript Object Notation) is a standard *schema-less* and *textual* serialization specification *inspired* by a subset [1] of the JavaScript programming language [47]. Douglas Crockford [2], currently a Distinguished Architect at PayPal, described the JSON serialization specification online [3] in 2002 and published the first draft of the JSON serialization specification in 2006 [40]. Douglas Crockford claims he *discovered* and *named* JSON, whilst Netscape was already using an unspecified variant as an interchange format as early as in 1996 in their web browser [41].

```
{
  "Image": {
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail": {
      "Url": "http://www.example.com/image/481989943",
      "Height": 125,
      "Width": 100
    },
    "Animated": false,
    "IDs": [ 116, 943, 234, 38793 ]
  }
}
```

Figure 3.1: A JSON document example taken from the official specification [21].

JSON is a human-readable open standard specification that consists of structures built on key-value pairs or list of ordered items. The data types it supports are objects, arrays, numbers, strings, `null`, and boolean constants `true` and `false`. A data structure encoded using JSON is referred to as a *JSON document*. [46] states that JSON documents are serialized as either UTF-8, UTF-16, or UTF-32 strings [38]. However, [21] mandate the use of UTF-8 for interoperability purposes. The serialization process involves recursively converting keys and values to strings and optionally removing meaningless new line, tab, and white space characters (a process known as *minification*) as shown in Figure 3.2.

```
{"Image":{"Width":800,"Height":600,"Title":"View from 15th Floor", "Thumbnail":{
"Url":"http://www.example.com/image/481989943","Height":125,"Width":100},
"Animated":false,"IDs":[116,943,234,38793]}}
```

Figure 3.2: A *minified* and semantically-equivalent version of the JSON document from Figure 3.1.

3.1.2 Relevance of JSON

JSON [46] is popular interchange specification in the context of cloud computing. In 2019, [147] found that JSON documents constitute a growing majority of the HTTP [55] responses served by Akamai, a leading Content Delivery Network (CDN) that serves 3 trillion HTTP requests daily. Gartner [4], a business insight research and advisory firm, forecasts that the cloud services market will grow 17% in 2020 to total $266.4 billion and that SaaS will remain the

[1] http://timelessrepo.com/json-isnt-a-javascript-subset
[2] https://www.crockford.com/
[3] https://www.json.org
[4] https://www.gartner.com

largest market segment [5]. SaaS systems typically provide *application programming interfaces* (APIs) and JSON was found to be the most common request and response format for APIs [6]. According to their study, JSON was used more than XML [104]. JSON popularity over XML can be attributed to the fact that in comparison to XML, JSON results in smaller bit-strings and in runtime and memory efficient serialization and deserialization implementations [101].

There is an on-going interest in JSON within the research community. [18] describe the first formal framework for JSON documents and introduce a query language for JSON documents named *JSON Navigational Logic* (JNL). There is a growing number of publications that discuss JSON in the context of APIs [157] [48] [50] and technologies that emerged from the JSON ecosystem such as the JSON Schema specification [111] [93] [68] [63]. Apart from cloud computing, JSON is relevant in areas such as databases [34] [84] [8] [62] [110], big data [161], analytics [103] [91], mobile applications [133] [66], 3D modelling [90], IoT [153] [100], biomedical research [81], and configuration files, for example [7]. [7] presents a high-level overview of the JSON ecosystem including a survey of popular schema languages and implementations, schema extraction technologies and novel parsing tools.

3.1.3 Shortcomings

Despite its popularity, JSON is neither a runtime-efficient nor a space-efficient serialization specification.

Runtime-efficiency. Serialization and deserialization often become a bottleneck in data-intensive, battery-powered, and resource-constrained systems. [103] state that big data applications may spend up to 90% of their execution time deserializing JSON documents, given that deserialization of textual specifications such as JSON is typically expensive using traditional state-machine-based parsing algorithms. [66] and [133] highlight the impact of serialization and deserialization speed on mobile battery consumption and resource-constrained mobile platforms. As a solution, [16] propose a promising JSON encoder and decoder that infers JSON usage patterns at runtime and self-optimizes by generating encoding and decoding machine code on the fly. Additionally, [88] propose a novel approach to efficiently parse JSON document by relying on SIMD processor instructions. [91] claim that applications parse entire JSON documents but typically only make use of certain fields. As a suggestion for optimization, they propose a lazy JSON parser that infers schemas for JSON documents at runtime and uses those schemas to speculate on the position of the fields that an application requires in order to avoid deserializing unnecessary areas of the JSON documents.

Space-efficiency. In comparison to JSON, [112] found that using a custom binary serialization specification reduced the overall network traffic by up to 94%. [15] conclude that JSON is not an appropriate specification for bandwidth-constrained communication systems citing the size of the documents as the main reason. [117] states that network traffic is one of the two biggest causes for battery consumption on mobile devices and therefore a space-efficient serialization specification could have a positive impact on energy savings.

There are several JSON-based specifications that highlight a need for compact JSON encodings:

- JSON Patch [25] is a specification for expressing a sequence of operations on JSON documents. [29] describe an algorithm called JDR to compute JSON Patch differences between two JSON documents optimized to keep the number of JSON Patch operations to a minimum for space-efficiency reasons.

- CityGML is an XML-based specification to represent 3D models of cities and landscapes. [90] introduce a JSON-based alternative to CityGML called CityJSON citing that CityGML documents are large enough that makes it difficult or even impossible to transmit and process them on the web. In comparison to CityGML, CityJSON results in smaller document size. However, the authors are looking for a binary JSON encoding to compress the CityJSON documents even further. Additionally, [141] explores how CityJSON documents can be compressed further using binary serialization specifications such as BSON [95] and CBOR [17].

- In the context of bioinformatics, *mmJSON* is a popular serialization specification used to encode representations of macromolecular structures. [19] introduce *MMTF*, a binary serialization specification to encode macromolecular structures based on MessagePack [64] to address the space-efficiency and runtime-efficiency concerns of using *mmJSON* to perform web-based structure visualization. In particular, using *mmJSON*, even after applying GZIP [44] compression, results in large macromolecular structures that are slow to download.

[5] https://www.gartner.com/en/newsroom/press-releases/2019-11-13-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2020

[6] https://www.programmableweb.com/news/json-clearly-king-api-data-formats-2020/research/2020/04/03

[7] https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

Due to their size, the largest macromolecular structures results in deserialization memory requirements that exceed the amount of memory typically available in web browsers.

3.2 JSON Schema

3.2.1 History and Relevance

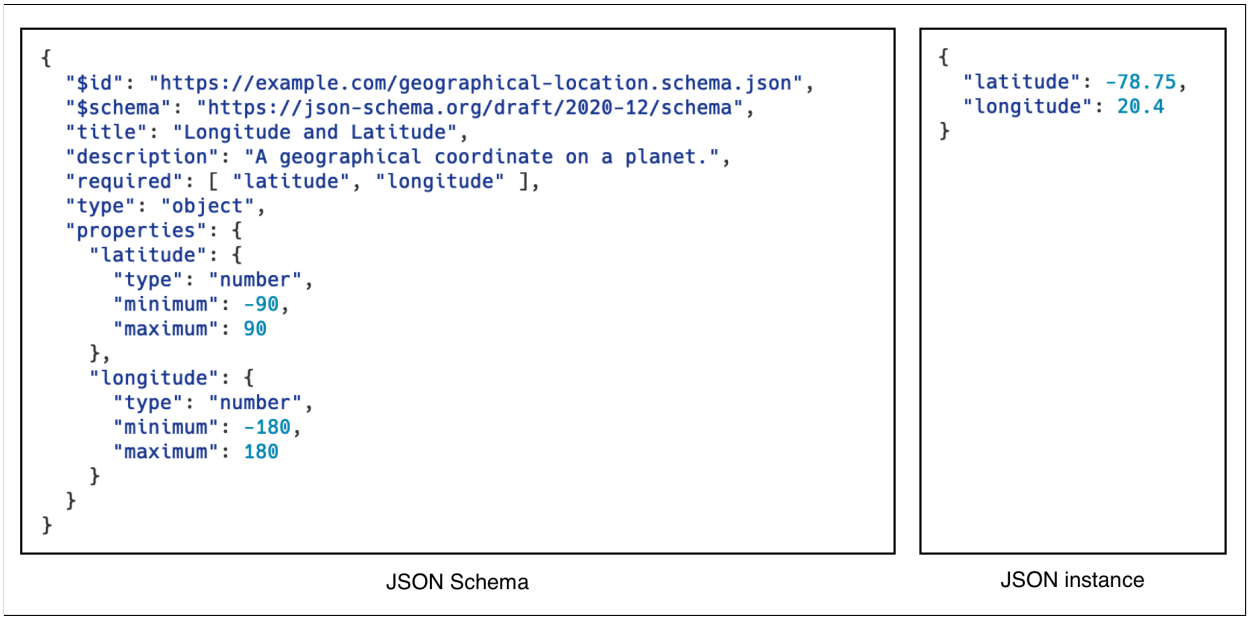

Figure 3.3: An example of a JSON Schema document adapted from the official website [8]. The JSON Schema 2020-12 [159] document (left) defines a JSON object with two required bounded numeric properties: *latitude* and *longitude*. The JSON [46] instance document (right) is an example of a JSON value that validates against the schema definition.

JSON Schema is a set of IETF specifications [159] [160] of a JSON-based format that defines the structure and meaning of JSON [46] documents for validation and annotation purposes.

JSON Schema was conceived at the AjaxWorld conference in 2007 during a session held by Douglas Crockford, the original specifier of the JSON [46] serialization specification. In that session, Kris Zyp, founder of Xucia [9] and technical writer at `json.com` [10] started a discussion on a schema language to validate JSON documents [11]. On that same year, Kris started collaborating with the OpenAjax Alliance [12] organization dedicated to the successful adoption of open and interoperable Ajax-based web technologies and wrote the first JSON Schema proposal [13] [14] based on the concepts of XML Schema [65], the RelaxNG [36] XML [104] schema language, and the Kwalify [15] open-source JSON [46] schema validator. In 2008, Kris launched the `json-schema.org` website [16] and started the JSON Schema Google Group [17] that is still active at the time of this writing. Since then, 9 versions of JSON Schema have been published at IETF. The latest version of JSON Schema, codenamed *2020-12*, was published on

[8] https://json-schema.org/learn/getting-started-step-by-step.html
[9] https://web.archive.org/web/20071027072650/http://xucia.com
[10] https://web.archive.org/web/20071026185558/http://www.json.com/author/kriszyp/
[11] https://web.archive.org/web/20071026190426/http://www.json.com/2007/09/27/json-schema-proposal-collaboration/
[12] http://www.openajax.org
[13] https://web.archive.org/web/20071026185150/http://json.com/json-schema-proposal/
[14] https://web.archive.org/web/20071027182021/http://www.json.com/2007/10/02/json-schema-proposal-rfc/
[15] https://web.archive.org/web/20080324120207/http://www.kuwata-lab.com/kwalify/
[16] https://web.archive.org/web/20080725083720/http://www.json-schema.org/
[17] https://groups.google.com/g/json-schema

December 2020 [159]. In early 2022, JSON Schema joined the OpenJS Foundation [18]. JSON Schema is currently an active open-source organization led by Ben Hutton. Ben Hutton is currently employed by Postman [19], a popular software company that provides API-related services.

At present, JSON Schema is a draft specification that has not obtained standard status. However, JSON Schema is the de-facto industry-standard schema language for JSON documents. As an example of JSON Schema popularity across industries, the NSA (National Security Agency) publishes security guidance for the use of JSON Schema [20]. In comparison to JSON Schema, other schema languages for JSON such as JSound [3], CUE [142], and JSON Type Definition [30] have not reached widespread use. The JSON Schema website [21] lists dozens of open-source JSON Schema implementations and related tooling written in more than 15 programming languages. As a notable example, in 2020, the AJV JSON Schema implementation was awarded a grant from Mozilla to sponsor its development [22]. During the same year, AJV also joined the OpenJS Foundation [23], as it has been identified as a key component in the web and JavaScript [47] ecosystems.

3.2.2 Use Cases

Due to the popularity of JSON [46], JSON Schema is adopted across industries. JSON Schema was originally designed to describe the structure of JSON documents for validation purposes. However, JSON Schema is now also used for innovative use cases beyond data validation such as API formal definitions [9] [157] [48] [126] [13] [107], UI generation [24] [25], databases [62] [84] [26] [27] [28] [29] [30] and code-generation [31] [32] [33] [34] [35] [36] [37] [38].

3.2.3 Characteristics

3.2.3.1 Expressiveness

Compared to other schema definition languages, the JSON Schema official specifications [159] [160] support a wide range of keywords to produce detailed data models. As noted by [111], [93] and [68], JSON Schema is a particularly expressive schema language. For example, JSON Schema supports arbitrarily nested and free-form objects and arrays, boolean logical operators, union data values, conditionals, external dependencies, property dependencies, and recursive schemas. Refer to Figure 3.4 for an example. In fact, JSON Schema is expressive enough to describe itself. A JSON Schema document that describes another JSON Schema document is referred to as a *meta-schema*. [111] provide a formal definition of the semantics and expressiveness of the now obsolete JSON Schema Draft 4 [51] and explore the theoretical aspects of the schema validation problem. They conclude that JSON Schema can represent a large set of problems as its operators can simulate finite state automata, simulate their containment, and simulate tree automata. More recent versions of JSON Schema introduced more advanced keywords that had not been considered in research yet. [93] conducted an empirical analysis of how JSON Schema is used in the industry and found that JSON Schema documents can become overly large (over 119K LoC), that recursion is not common in practice outside

[18] https://openjsf.org/blog/2022/01/31/json-schema-joins-openjs-foundation/
[19] https://blog.postman.com/ben-hutton-joins-postman-to-lead-json-schema-strategy/
[20] https://apps.nsa.gov/iaarchive/library/reports/security_guidance_for_json.cfm
[21] https://json-schema.org/implementations.html
[22] https://ajv.js.org/news/2020-08-14-mozilla-grant-openjs-foundation.html
[23] https://openjsf.org/blog/2020/08/14/ajv-joins-openjs-foundation-as-an-incubation-project/
[24] https://github.com/rjsf-team/react-jsonschema-form
[25] https://github.com/ui-schema/ui-schema
[26] https://better.engineering/jsonschema2db/
[27] https://github.com/gavinwahl/postgres-json-schema
[28] https://docs.mongodb.com/manual/core/schema-validation/
[29] https://docs.mongodb.com/manual/reference/operator/query/jsonSchema/
[30] https://github.com/mbroadst/thinkagain
[31] https://github.com/YousefED/typescript-json-schema
[32] https://github.com/victools/jsonschema-generator
[33] https://github.com/andyglow/scala-jsonschema
[34] https://github.com/dragonwasrobot/json-schema-to-elm
[35] https://github.com/pwall567/json-kotlin-schema-codegen
[36] https://github.com/swaggest/php-code-builder
[37] https://github.com/Marwes/schemafy/
[38] https://statham-schema.readthedocs.io
[39] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/ref.json#L273-L360

```
{
  "$id": "http://example.com/tree",
  "description": "tree of nodes",
  "type": "object",
  "properties": {
    "meta": { "type": "string" },
    "nodes": {
      "type": "array",
      "items": { "$ref": "node" }
    }
  },
  "required": [ "meta", "nodes" ],
  "$defs": {
    "node": {
      "$id": "http://example.com/node",
      "description": "node",
      "type": "object",
      "properties": {
        "value": { "type": "number" },
        "subtree": { "$ref": "tree" }
      },
      "required": [ "value" ]
    }
  }
}
```

JSON Schema

```
{
  "meta": "root",
  "nodes": [
    {
      "value": 1,
      "subtree": {
        "meta": "child",
        "nodes": [
          { "value": 1.1 },
          {
            "value": 1.2,
            "subtree": {
              "meta": "subchild",
              "nodes": [
                { "value": 3.1 }
              ]
            }
          }
        ]
      }
    },
    {
      "value": 2,
      "subtree": {
        "meta": "child",
        "nodes": [
          { "value": 2.1 },
          { "value": 2.2 }
        ]
      }
    }
  ]
}
```

Matching JSON Instance

Figure 3.4: On the left, a JSON Schema [159] definition adapted from the official test suite [39] that uses recursive references between schemas to validate arbitrary N-ary tree data structures consisting of real numbers. On the right, an example matching instance.

the definition of meta-schemas and that most schemas are open-ended and allow the presence of extra properties not declared in the schema.

In order to mitigate the complexity of JSON Schema for newcomers, the JSON Schema specification lead mentions the idea of defining the smallest possible subset of JSON Schema that caters to the most common use cases based on data collected by popular implementations of JSON Schema [132].

3.2.3.2 Flexibility

The keywords supported by the JSON Schema specifications [159] [160] are used to annotate or impose constrains over the instances that validate against the given schema. The JSON Schema specification does not make any of its keywords required. Therefore, every possible constrain supported by JSON Schema is opt-in as exemplified in Figure 3.5. As a consequence, JSON Schema definitions may be strict or loose depending on the schema author and the problem domain. For example, a JSON Schema definition may meticulously describe each property of its instances and provide an extensive amount of constrains. Or as an extreme example, a JSON Schema definition may impose no constrains and behave like the wildcard schema that validates every JSON [46] document. For this reason, the JSON Schema specification lead describes JSON Schema as a *constrains* language rather than as a modelling language [132]. This flexibility can be used to write schema definitions that follow a hybrid approach: provide detailed models of certain parts of the data structure, while leaving other parts of the data structure open to free-form extension.

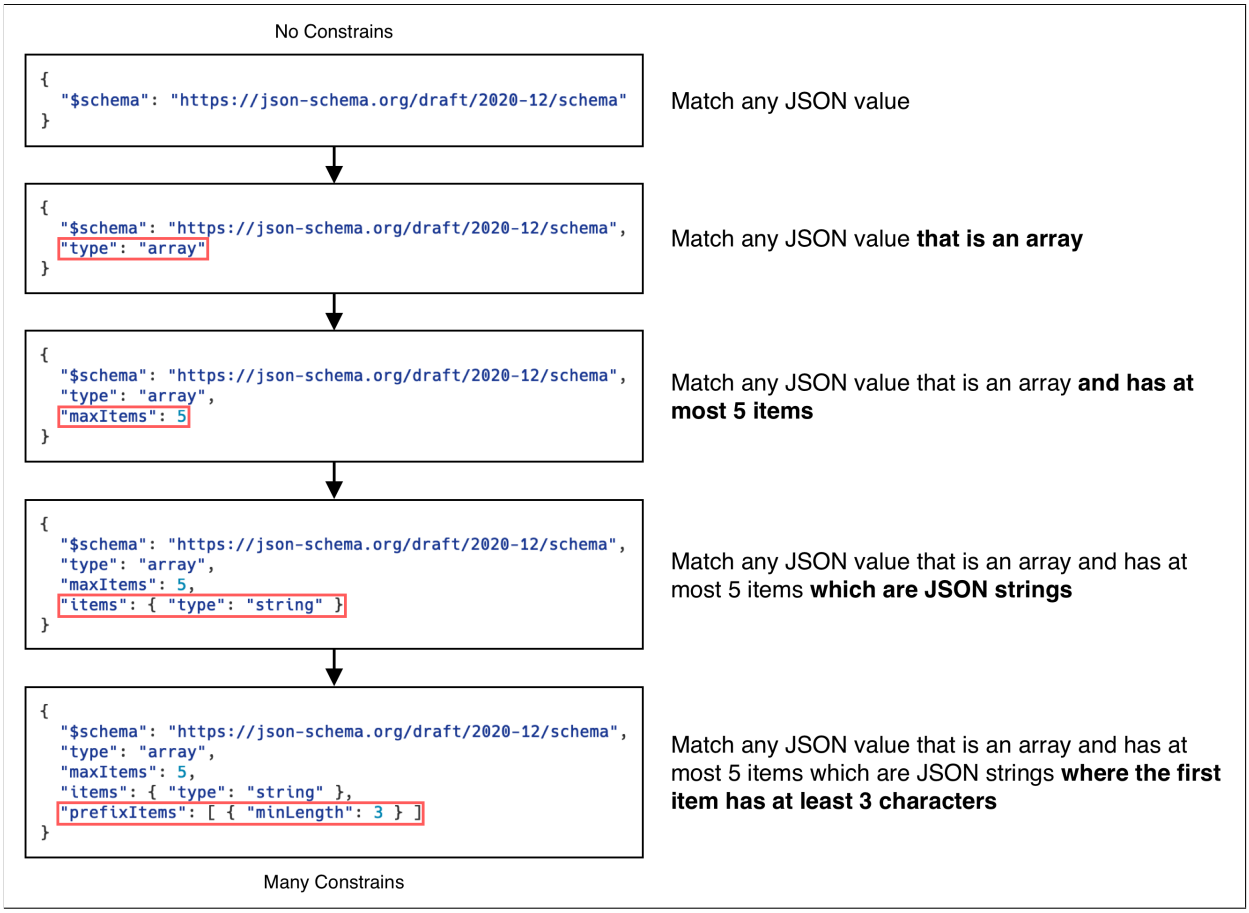

Figure 3.5: A series of JSON Schema definitions where each subsequent definition adds an additional keyword highlighted in red. The presence of each additional keyword imposes one further constrain to the set of matching instances as described in the right.

3.2.3.3 Extensibility

The JSON Schema format has been designed with extensibility in mind. The JSON Schema Core specification [159] describes the foundational keywords that must be supported by any JSON Schema implementation and introduces a mechanism called *vocabularies*, through which implementations can define and use additional groups of keywords as exemplified in Figure 3.6. In fact, JSON Schema itself makes use of this mechanism to divide the set of keywords it defines on the official specifications [159] [160] into logical groups that can be mixed and matched by the schema author. Additionally, the concept of a meta-schema, a JSON Schema that validates other JSON Schema definitions, can be used to impose arbitrary restrictions on schema definitions. For example, a meta-schema may declare that a JSON Schema can only describe and validate numeric data structures.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/meta/general-use-example",
  "$dynamicAnchor": "meta",
  "$vocabulary": {
    "https://json-schema.org/draft/2020-12/vocab/core": true,
    "https://json-schema.org/draft/2020-12/vocab/applicator": true,
    "https://json-schema.org/draft/2020-12/vocab/validation": true,
    "https://example.com/vocab/example-vocab": true
  },
  "allOf": [
    { "$ref": "https://json-schema.org/draft/2020-12/meta/core" },
    { "$ref": "https://json-schema.org/draft/2020-12/meta/applicator" },
    { "$ref": "https://json-schema.org/draft/2020-12/meta/validation" },
    { "$ref": "https://example.com/meta/example-vocab" }
  ]
}
```

Figure 3.6: An example of extending JSON Schema with a custom vocabulary adapted from [159]. This JSON Schema meta-schema definition enables instances to use the keywords defined by the official Core, Applicator and Validation vocabularies introduced in [159] and [160]. Additionally, it enables instances to use the keywords defined by the fictitious vocabulary `https://example.com/vocab/example-vocab`. The meta-schema validates the keywords introduced by each declared vocabulary by stating that instances must successfully validate against the meta-schema of every declared vocabulary.

4 | A Taxonomy JSON Documents

4.1 The SchemaStore Dataset

SchemaStore [1] is an Apache-2.0-licensed [2] open-source collection of over 300 JSON Schema [158] documents which describe popular JSON-based [46] formats such as CityJSON [141] and JSON Patch [25]. The SchemaStore API can be integrated with code editors to offer auto-completion and validation when writing JSON documents. The SchemaStore project was started by Mads Kristensen [3] in 2014 while working as a Senior Program Manager focused on the Visual Studio IDE at Microsoft. For the purpose of benchmarking, I will make use of SchemaStore's extensive test suite which consists of over 400 real-world JSON documents matching the respective schemas [4]. This paper refers to the commit hash `0b6bd2a08005e6f7a65a68acaf3064d6e2670872` of the SchemaStore repository hosted on GitHub [5]. I believe that the SchemaStore test suite is a good representation of the set of JSON documents used across industries.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Apple Universal Link, App Site Association",
  "type": "object",
  "properties": {
    "applinks": {
      "type": "object",
      "properties": {
        "apps": {
          "enum": [ [] ],
          "description": "Must be an empty array"
        },
        "details": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "appID": {
                "type": "string",
                "description":
                  "The value of the appID key is the team ID or app ID prefix, followed by the bundle ID"
              },
              "paths": {
                "type": "array",
                "items": {
                  "type": "string",
                  "description":
                    "Ordered list of paths to open in the mobile app. Prefix with 'NOT ' to remove a path"
                }
              }
            }
          }
        }
      },
      "required": [ "apps", "details" ]
    }
  },
  "required": [ "applinks" ]
}
```

Figure 4.1: An example JSON Schema Draft 4 [51] document from SchemaStore [6] that describes an Apple Associated Domain file [7] to associate an iOS app and a website.

[1] https://www.schemastore.org
[2] http://www.apache.org/licenses/LICENSE-2.0.html
[3] https://github.com/madskristensen
[4] https://github.com/SchemaStore/schemastore/tree/master/src/test
[5] https://github.com/SchemaStore/schemastore
[6] https://github.com/SchemaStore/schemastore/blob/0b6bd2a08005e6f7a65a68acaf3064d6e2670872/src/schemas/json/apple-app-site-association.json
[7] https://developer.apple.com/documentation/safariservices/supporting_associated_domains
[8] https://github.com/SchemaStore/schemastore/blob/0b6bd2a08005e6f7a65a68acaf3064d6e2670872/src/test/apple-app-site-association/apple-app-site-association_getting-started.json

```
{
  "applinks": {
    "apps": [],
    "details": [
      {
        "appID": "9JA89QQLNQ.com.apple.wwdc",
        "paths": [ "/wwdc/news/", "/videos/wwdc/2015/*" ]
      },
      {
        "appID": "ABCD1234.com.apple.wwdc",
        "paths": [ "*" ]
      }
    ]
  }
}
```

Figure 4.2: An example JSON [46] document that matches the schema definition from Figure 4.1 taken from SchemaStore's test suite [8].

4.2 Taxonomy Definition

Serializing two data structures that match the same schema definition but consist of different values is likely to result in similar byte sizes. However, serializating two data structures with the same values but different structures may produce diverse results, even when utilising the same serialization specification. Therefore, I conclude that the structure and the type of content affects the size of the serialized bit-strings more than the actual values. Under this assumption, to produce a representative size benchmark, it is essential to measure binary serialization specifications using a set of JSON [46] documents that differ in structure, type of content and size.

To solve the input data selection problem, it is required to have a process to categorize JSON [46] documents depending on such characteristics. In this way, I present a taxonomy consisting of 36 categories listed in Table 4.1. The taxonomy qualifies JSON documents based on their size, type of content, nesting, structural, and redundancy characteristics. While most JSON documents in practice are objects or arrays, this taxonomy is also applicable to JSON documents consisting of single scalar values and strings. I hope that this taxonomy forms a common basis to talk about JSON documents in a high-level manner beyond the benchmarking problem.

4.2.1 Size

In order to categorize JSON documents in a sensible manner using a small set of size categories, I first calculate the byte-size distribution of the JSON documents in the SchemaStore test suite introduced in section 4.1. The results are illustrated in Figure 4.4. Based on these results, I group JSON documents into three categories:

- **Tier 1 Minified $<$ 100 bytes.** A JSON document is in this category if its UTF-8 [38] *minified* form occupies less than 100 bytes [151].
- **Tier 2 Minified $\geq$ 100 $<$ 1000 bytes.** A JSON document is in this category if its UTF-8 [38] *minified* form occupies 100 bytes or more, but less than 1000 bytes [151].
- **Tier 3 Minified $\geq$ 1000 bytes.** A JSON document is in this category if its UTF-8 [38] *minified* form occupies 1000 bytes or more [151].

```
$ node
Welcome to Node.js v12.22.1.
Type ".help" for more information.
> const document = { foo: 'bar' }
undefined
> Buffer.byteLength(JSON.stringify(document), 'utf8')
13
```

Figure 4.3: The UTF-8 [38] byte-size of a JSON document in *minified* form can be determined using a Node.js [9] interactive REPL session by combining the `JSON.stringify` and the `Buffer.byteLength` functions as demonstrated in this figure. In this example, I determine the size of the JSON document `{ "foo": "bar" }` to be 13 bytes.

4.2.2 Content Type

The taxonomy categorises a JSON document based on the data types that dominate its content. I calculate this characteristic based on the number of values of a certain data type that a JSON document contains and the byte-size that these data values occupy in the serialized bit-string. I take both of these measures into account as serializing many small instances result in more metadata overhead than serializing a few large instances for a given type.

[9] https://nodejs.org/

[10] https://github.com/SchemaStore/schemastore/blob/0b6bd2a08005e6f7a65a68acaf3064d6e2670872/src/test/sarif/BinSkim.AllRules.sarif.json

The JSON [46] serialization specification supports the following data types: *object*, *array*, *boolean*, *string*, *number*, and *null*. I consider objects and arrays to represent *structural* values, strings to represent *textual* values, and numbers to represent *numeric* values. For simplicity, I consider *true*, *false*, and *null* to represent *boolean* values as in three-valued logic [116]. I use this data type categorization to define the *textual weight*, *numeric weight*, and *boolean weight* for a given JSON document.

The weight metrics for a JSON document are based on a common formula where C is the total number of data values in the JSON document and S is the total byte-size of the JSON document in *minified* form [151]:

Table 4.1: There are 36 categories defined in my JSON documents taxonomy. The second column contains acronyms for each category name. In terms of size, a JSON document can either be *Tier 1 Minified* $<$ *100 bytes*, *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes*, or *Tier 3 Minified* $\geq$ *1000 bytes*. In terms of content, a JSON document can either be *numeric*, *textual*, or *boolean*. Finally, in terms of structure, a JSON document can either be *flat* or *nested*.

Category				**Acronym**
Tier 1 Minified $<$ 100 bytes	Numeric	Redundant	Flat	`Tier 1 NRF`
Tier 1 Minified $<$ 100 bytes	Numeric	Redundant	Nested	`Tier 1 NRN`
Tier 1 Minified $<$ 100 bytes	Numeric	Non-Redundant	Flat	`Tier 1 NNF`
Tier 1 Minified $<$ 100 bytes	Numeric	Non-Redundant	Nested	`Tier 1 NNN`
Tier 1 Minified $<$ 100 bytes	Textual	Redundant	Flat	`Tier 1 TRF`
Tier 1 Minified $<$ 100 bytes	Textual	Redundant	Nested	`Tier 1 TRN`
Tier 1 Minified $<$ 100 bytes	Textual	Non-Redundant	Flat	`Tier 1 TNF`
Tier 1 Minified $<$ 100 bytes	Textual	Non-Redundant	Nested	`Tier 1 TNN`
Tier 1 Minified $<$ 100 bytes	Boolean	Redundant	Flat	`Tier 1 BRF`
Tier 1 Minified $<$ 100 bytes	Boolean	Redundant	Nested	`Tier 1 BRN`
Tier 1 Minified $<$ 100 bytes	Boolean	Non-Redundant	Flat	`Tier 1 BNF`
Tier 1 Minified $<$ 100 bytes	Boolean	Non-Redundant	Nested	`Tier 1 BNN`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Numeric	Redundant	Flat	`Tier 2 NRF`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Numeric	Redundant	Nested	`Tier 2 NRN`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Numeric	Non-Redundant	Flat	`Tier 2 NNF`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Numeric	Non-Redundant	Nested	`Tier 2 NNN`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Textual	Redundant	Flat	`Tier 2 TRF`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Textual	Redundant	Nested	`Tier 2 TRN`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Textual	Non-Redundant	Flat	`Tier 2 TNF`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Textual	Non-Redundant	Nested	`Tier 2 TNN`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Boolean	Redundant	Flat	`Tier 2 BRF`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Boolean	Redundant	Nested	`Tier 2 BRN`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Boolean	Non-Redundant	Flat	`Tier 2 BNF`
Tier 2 Minified $\geq$ 100 $<$ 1000 bytes	Boolean	Non-Redundant	Nested	`Tier 2 BNN`
Tier 3 Minified $\geq$ 1000 bytes	Numeric	Redundant	Flat	`Tier 3 NRF`
Tier 3 Minified $\geq$ 1000 bytes	Numeric	Redundant	Nested	`Tier 3 NRN`
Tier 3 Minified $\geq$ 1000 bytes	Numeric	Non-Redundant	Flat	`Tier 3 NNF`
Tier 3 Minified $\geq$ 1000 bytes	Numeric	Non-Redundant	Nested	`Tier 3 NNN`
Tier 3 Minified $\geq$ 1000 bytes	Textual	Redundant	Flat	`Tier 3 TRF`
Tier 3 Minified $\geq$ 1000 bytes	Textual	Redundant	Nested	`Tier 3 TRN`
Tier 3 Minified $\geq$ 1000 bytes	Textual	Non-Redundant	Flat	`Tier 3 TNF`
Tier 3 Minified $\geq$ 1000 bytes	Textual	Non-Redundant	Nested	`Tier 3 TNN`
Tier 3 Minified $\geq$ 1000 bytes	Boolean	Redundant	Flat	`Tier 3 BRF`
Tier 3 Minified $\geq$ 1000 bytes	Boolean	Redundant	Nested	`Tier 3 BRN`
Tier 3 Minified $\geq$ 1000 bytes	Boolean	Non-Redundant	Flat	`Tier 3 BNF`
Tier 3 Minified $\geq$ 1000 bytes	Boolean	Non-Redundant	Nested	`Tier 3 BNN`

$$\frac{\frac{K \times 100}{C} \times \frac{B \times 100}{S}}{100} \tag{4.1}$$

- **Textual Weight.** In this case, K is the number of string values in the JSON document and B is the cummulative byte-size occupied by the string values in the JSON document. The double quotes surrouding string values are considered part of the byte-size occupied by a string. Therefore, a string value encoded in a UTF-8 [38] JSON document occupies at least $2 + N$ bytes where N corresponds to number of code-points in the string.
- **Numeric Weight.** In this case, K is the number of numeric values in the JSON document and B is the cummulative byte-size occupied by the numeric values in the JSON document. Each numeric digit and auxiliary characters such as the minus sign $(-)$ and the period $(.)$ for representing real numbers count towards the byte-size of the numeric value.
- **Boolean Weight.** In this case, K is the number of boolean values in the JSON document and B is the cummulative byte-size occupied by the boolean values in the JSON document. The UTF-8 [38] JSON encoding represents *true* using 4 bytes, *false* using 5 bytes, and *null* using 4 bytes.

I rely on the previous weight definitions to provide the content type taxonomy for JSON documents based on whether they are *textual*, *numeric*, or *boolean*. Given an input JSON document, consider W_t, W_n, and W_b to represent its textual, numeric and boolean weights, respectively:

- **Textual.** A JSON document is textual if $W_t \geq W_n \geq W_b$.
- **Numeric.** A JSON document is numeric if $W_n \geq W_t \geq W_b$.
- **Boolean.** A JSON document is numeric if $W_b \geq W_t \geq W_n$.

If two or more of the content type weight values are equal and greater than the rest, such JSON document is considered to hold more than one type of content qualifier. For example, if $W_t = W_n$ and $W_t > W_b$, then the JSON document is equally considered *textual* and *numeric*.

The results of executing this aspect of the taxonomy on the SchemaStore test suite introduced in section 4.1 are shown in Figure 4.5.

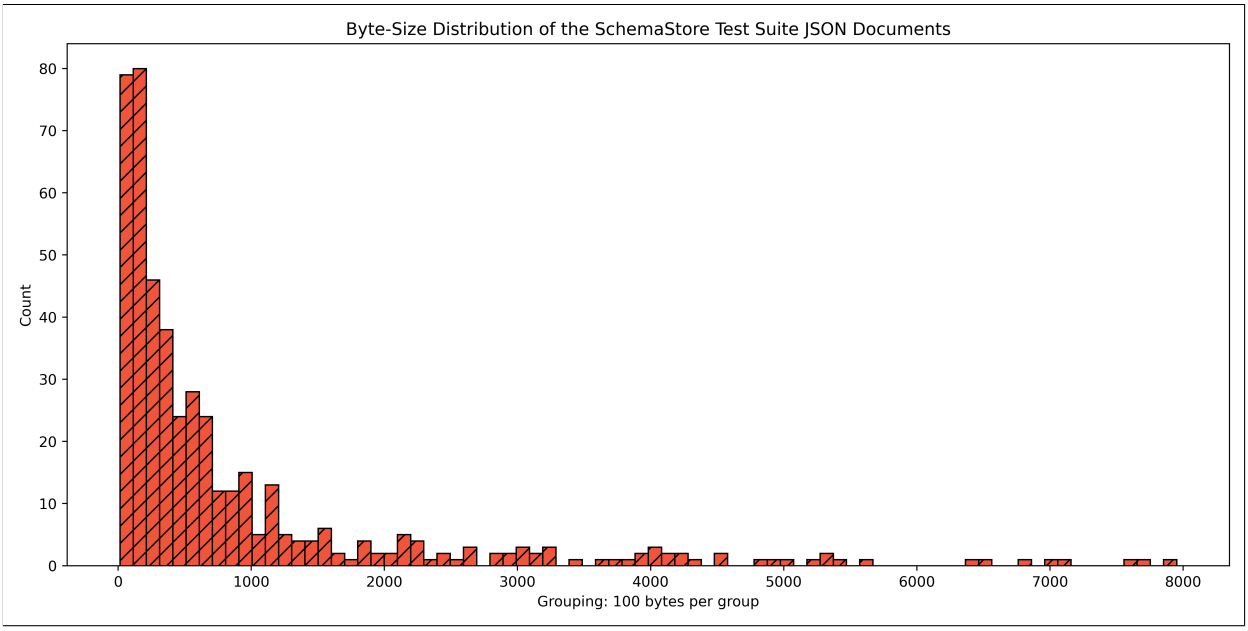

Figure 4.4: The byte-size distribution (in 100 byte groups up to 8000 bytes for illustration purposes) of the 480 JSON documents present in the SchemaStore test suite introduced in section 4.1. Most JSON documents weigh less than 1000 bytes. The largest JSON document weighs 545392 bytes [10].

4.2.3 Redundancy

The taxonomy measures redundancy as the percentage of values in a given JSON document that are duplicated taking scalar and composite data types into account.

In comparison to schema-less serialization formats, schema-driven serialization formats make use of schema definitions to avoid encoding object keys. This taxonomy is designed to aid in categorizing JSON documents based on characteristics that impact data serialization. For these reasons, the number of duplicated object keys in the redundancy metric is irrelevant for the schema-driven subset of the selection of binary serializion formats and is not taken into account.

Let *JSON* be the set of JSON documents as defined in the data model introduced by [18] with the exception that the $[*]$ object operator results in a *sequence*, instead of a set, of values in the given JSON object. Consider a new $[\&]$ operator defined using the Z formal specification notation [78] that results in the flattened sequence of atomic and compositional structure values of the given JSON document:

$$
\begin{array}{|l}
\_[\&] : JSON \rightarrow \mathrm{seq}\, JSON \\
\hline
\forall J : JSON \bullet \\
\quad J[\&] = \langle J \rangle \frown J[*][0][\&] \frown \ldots \frown J[*][\#J][\&] \qquad \text{if } J \text{ is an object} \\
\quad J[\&] = \langle J \rangle \frown J_0[\&] \frown J'[\&] \qquad \text{if } J \text{ is an array} \\
\quad J[\&] = \langle J \rangle \qquad \text{otherwise}
\end{array}
$$

Using this operator, R_J is defined as the percentage of duplicate values in the JSON document J:

$$R_J = \frac{(\#J[\&] - \#\{v \mid v \; in \; J[\&]\}) \times 100}{\#J[\&]} \tag{4.2}$$

In order to categorize JSON documents in a sensible manner, the taxonomy distinguishes between *redundant* JSON documents and *non-redundant* JSON documents. The redundancy distribution of the JSON documents in the SchemaStore test suite introduced in section 4.1 is computed in Figure 4.6. Using these results, this taxonomy aspect is defined as follows:

- **Redundant.** A JSON document J is redundant if $R_J \geq 25\%$
- **Non-Redundant.** A JSON document J is redundant if $R_J < 25\%$

[11] https://github.com/SchemaStore/schemastore/blob/0b6bd2a08005e6f7a65a68acaf3064d6e2670872/src/

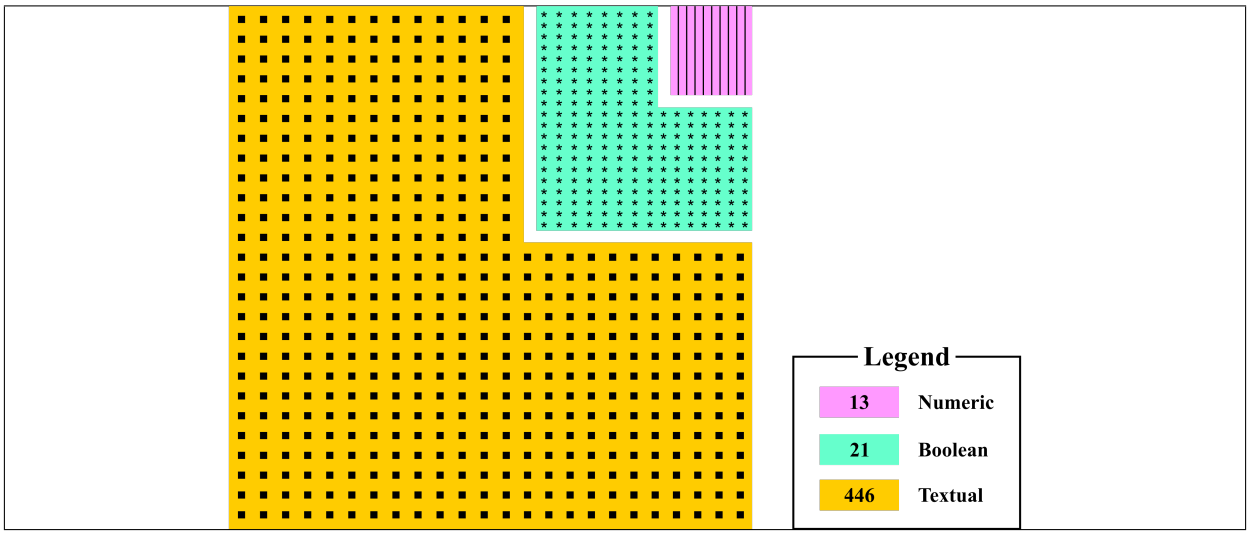

Figure 4.5: Out of the 480 JSON documents in the SchemaStore test suite introduced in section 4.1, 446 are textual, 21 are boolean, and 13 are numeric.

4.2.4 Structure

[18] propose that connected acyclic undirected graphs which resembles a tree structure are a natural representation for JSON documents as exemplified in Figure 4.7. I use the following definitions for two features associated with trees: *height* and *level*.

Definition 1. *The height of a node is the number of edges on the longest downward path between that node and a leaf. The height of a tree is the height of its root.*

Definition 2. *The level of a node is defined by 1 + the number of connections between the node and the root. The*

`test/csslintrc/WebAnalyzer.json`

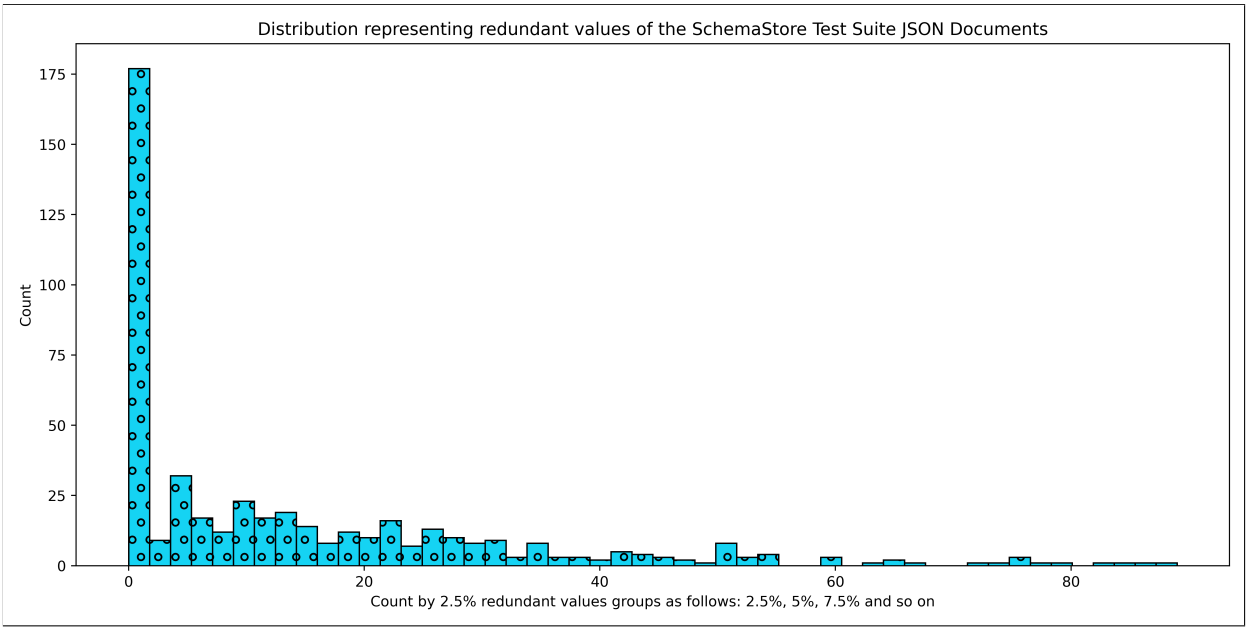

Figure 4.6: Distribution representing redundant values. First, I calculate the percentage of redundant values of the 480 JSON documents present in the SchemaStore test suite introduced in section 4.1 and second, I count them by 2.5% redundant values groups as follows: 2.5%, 5%, 7.5% and so on. Most JSON documents are strictly non-redundant. However there are instances of almost every 2.5% redundancy groups in the plot. The most redundant JSON document has a value redundancy of 88.8% [11].

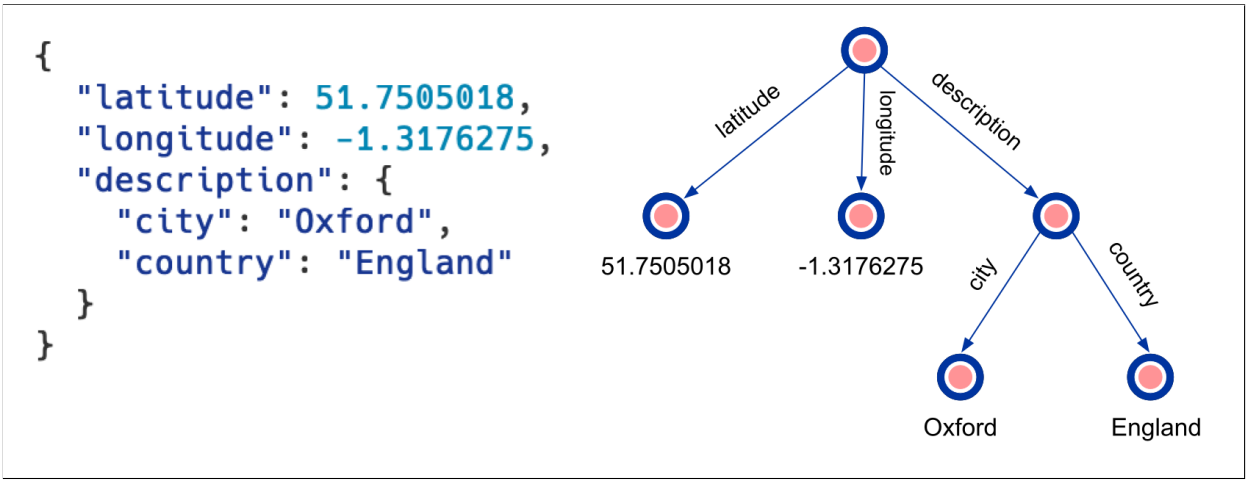

Figure 4.7: An example JSON document and its corresponding connected acyclic undirected graph representation.

level is depth + 1.

Using the definition of height, I extrapolate that the height of the tree determines the height of a given JSON document. Using the definition of level, I extrapolate that the *size* of a level in the tree equals the sum of the byte-size of every textual, numeric, and boolean values whose nodes have the corresponding level. Therefore, the *largest level* is the level with the highest size without taking into account the subtree at depth 0.

The *nesting weight* of a JSON document J, referred to as N_J, is defined as the product of its height and largest level minus 1. I do not consider the byte-size overhead introduced by compositional structures (object and array) in the JSON document as I found that it is highly correlated to its nesting characteristics.

$$N_J = \text{height} \times \text{largest level} - 1 \tag{4.3}$$

In order to categorize JSON documents in a sensible manner, the taxonomy distinguishes between *flat* JSON documents and *nested* JSON documents. The nesting weight distribution of the JSON documents in the SchemaStore test suite introduced in section 4.1 is computed in Figure 4.8. Using those results, this taxonomy aspect is defined as follows:

- **Flat.** A JSON document J is *flat* if N_J less than the empirically-derived threshold integer value 10.
- **Nested.** A JSON document J is *nested* if N_J is greater than or equal to the empirically-derived threshold integer value 10.

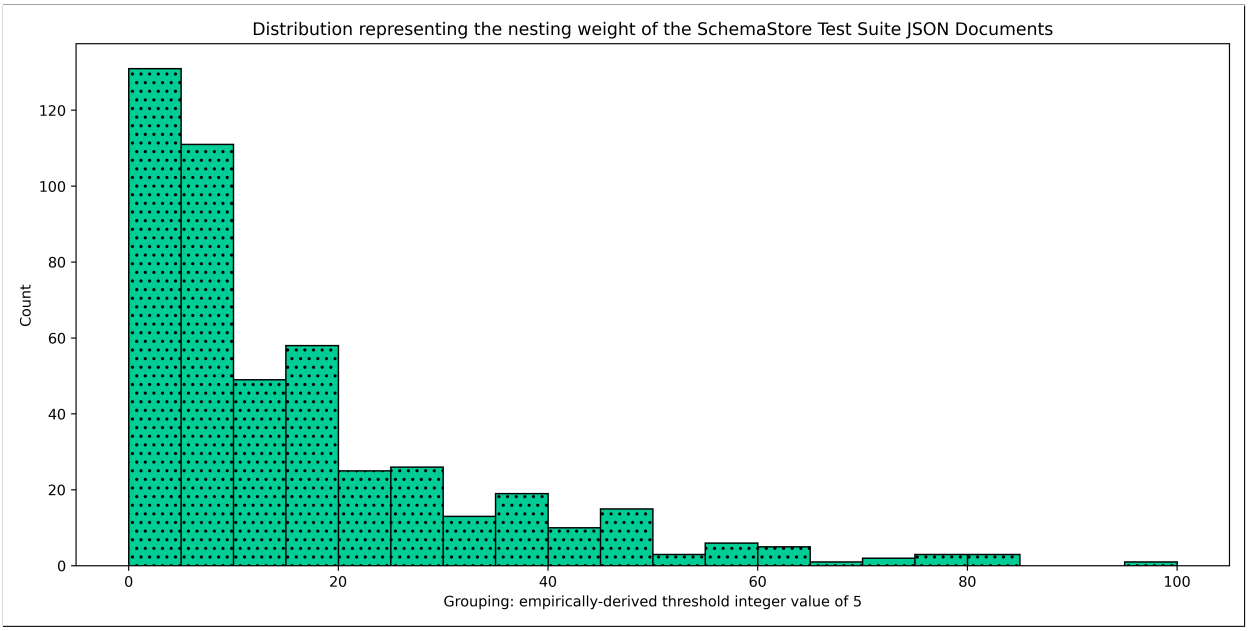

Figure 4.8: The nesting weight distribution of the 480 JSON documents present in the SchemaStore test suite introduced in section 4.1 grouped by the empirically-derived threshold 5. Most JSON documents have a nesting weight of under 20. However, there are JSON documents with a nesting weight of up to 100 [12].

4.2.5 Demonstration

To demonstrate my conceptual taxonomy, I apply it to the JSON document listed in Figure 4.9. Table 4.2 provides a breakdown of this JSON document including information such as the number of edges and byte size of every valid JSON Pointer path [26]. This document is a *Tier 2 Minified* $\geq 100 < 1000$ *bytes*, *numeric*, *non-redundant*, and *nested* (SNNN according to Table 4.1) JSON document:

[12] https://github.com/SchemaStore/schemastore/blob/0b6bd2a08005e6f7a65a68acaf3064d6e2670872/src/test/cloudify/utilities-cloudinit-simple.json

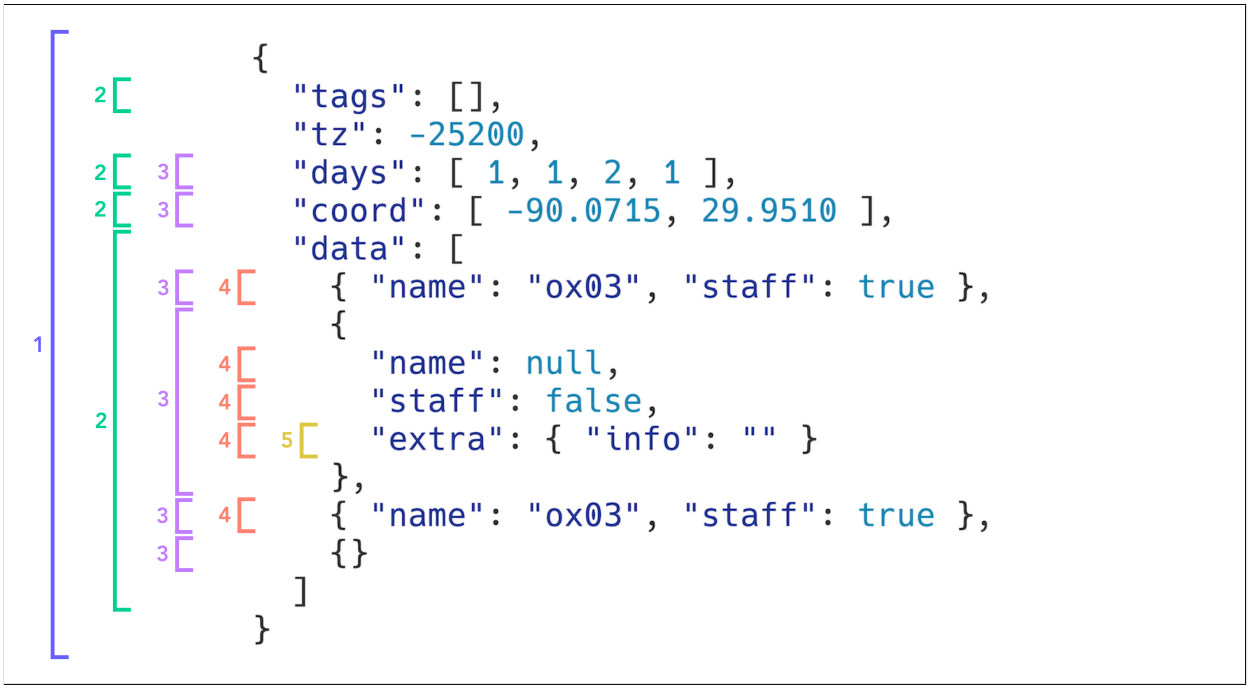

Figure 4.9: An example *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes, numeric, non-redundant, and nested* JSON document taken from my previous work [151]. The annotations at the left highlight each level in the JSON document. The height of this document is $(5 - 1 = 4)$ (the highest level minus 1).

- **Tier 2 Minified $\geq$ 100 $<$ 1000 bytes.** The size of the JSON document is 184 bytes. 184 is greater than 100 but less than 1000, therefore the JSON document from Figure 4.9 is *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes* according to the taxonomy.
- **Numeric.** Table 4.2 shows that the JSON document has 24 values corresponding to its set of valid JSON Pointer [26] paths. Of those, 7 (29.16%) are numeric, 3 (12.5%) are textual, and 4 (16.66%) are boolean. Out of the 184 total bytes from the JSON document, 24 bytes (13.04%) correspond to numeric values, 14 bytes (7.60%) correspond to textual values, and 17 bytes (9.23%) correspond to boolean values. The numeric weight is $29.16 \times 13.04/100 = 3.80$, the textual weight is $12.5 \times 7.60/100 = 0.95$, and the boolean weight is $16.66 \times 9.23/100 = 1.53$. 3.80 is greater than 0.95 and 1.53, therefore the JSON document from Figure 4.9 is *numeric* according to the taxonomy.
- **Non-Redundant.** The JSON document consists of 24 values. Out of those, the numeric value 1 appears in the JSON Pointer [26] paths `/days/0`, `/days/1`, and `/days/2`. The textual value *ox03* appears at `/data/0/name` and `/data/2/name`. Similarly, the boolean value *true* appears at `/data/0/staff` and `/data/2/staff`. Furthermore, the objects `/data/0` and `/data/2` are equal. Therefore, only 19 out of the 24 values in the JSON document are unique. I conclude that only 5 (20.83%) of its values are redundant, so the JSON document from Figure 4.9 is *non-redundant* according to the taxonomy.
- **Nested.** The height is 4, awarded to the pointer `/data/1/extra/info`. I calculate the byte-size of each level by adding the byte-size of each non-structural value in such level. Level 2 occupies 6 bytes, level 3 occupies 18 bytes, level 4 occupies 29 bytes and level 5 occupies 2 bytes. Therefore, level 4 is the largest level. The nesting weight of the JSON document is $4 \times (4 - 1) = 12$ (the height multiplied by the largest level minus 1). 12 is greater than 10, therefore the JSON document from Figure 4.9 is *nested* according to the taxonomy.

4.2.6 JSON Stats Analyzer

I built and published a free-to-use online tool at `https://www.jsonbinpack.org/stats/` to automatically categorize JSON documents according to the taxonomy defined in this section and provide summary statistics. Fig-

Table 4.2: A breakdown of the JSON document from Figure 4.9 in terms of its valid JSON Pointer [26] paths, value type, level , byte-size, and redundancy.

JSON Pointer	Type	Level	Byte-size	Same As
`/`	Structural	1	184	
`/tags`	Structural	2	2	
`/tz`	Numeric	2	6	
`/days`	Structural	2	9	
`/days/0`	Numeric	3	1	`/days/1`, `/days/3`
`/days/1`	Numeric	3	1	`/days/0`, `/days/3`
`/days/2`	Numeric	3	1	
`/days/3`	Numeric	3	1	`/days/0`, `/days/1`
`/coord`	Structural	2	17	
`/coord/0`	Numeric	3	8	
`/coord/1`	Numeric	3	6	
`/data`	Structural	2	110	
`/data/0`	Structural	3	28	`/data/2`
`/data/0/name`	Textual	4	6	`/data/2/name`
`/data/0/staff`	Boolean	4	4	`/data/2/staff`
`/data/1`	Structural	3	47	
`/data/1/name`	Boolean	4	4	
`/data/1/staff`	Boolean	4	5	
`/data/1/extra`	Structural	4	11	
`/data/1/extra/info`	Textual	5	2	
`/data/2`	Structural	3	28	`/data/0`
`/data/2/name`	Textual	4	6	`/data/0/name`
`/data/2/staff`	Boolean	4	4	`/data/0/staff`
`/data/3`	Structural	3	2	

ure 4.10 demonstrates the summary statistics analyzed for the *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes, numeric, non-redundant, and nested* JSON document from Figure 4.9.

The tool is developed using the TypeScript [13] programming language, the CodeMirror [14] open-source embeddable web editor, and the Tailwind CSS [15] open-source web component framework. The web application is deployed to the GitHub Pages [16] free static-hosting service.

[13] https://www.typescriptlang.org
[14] https://codemirror.net
[15] https://tailwindcss.com
[16] https://pages.github.com

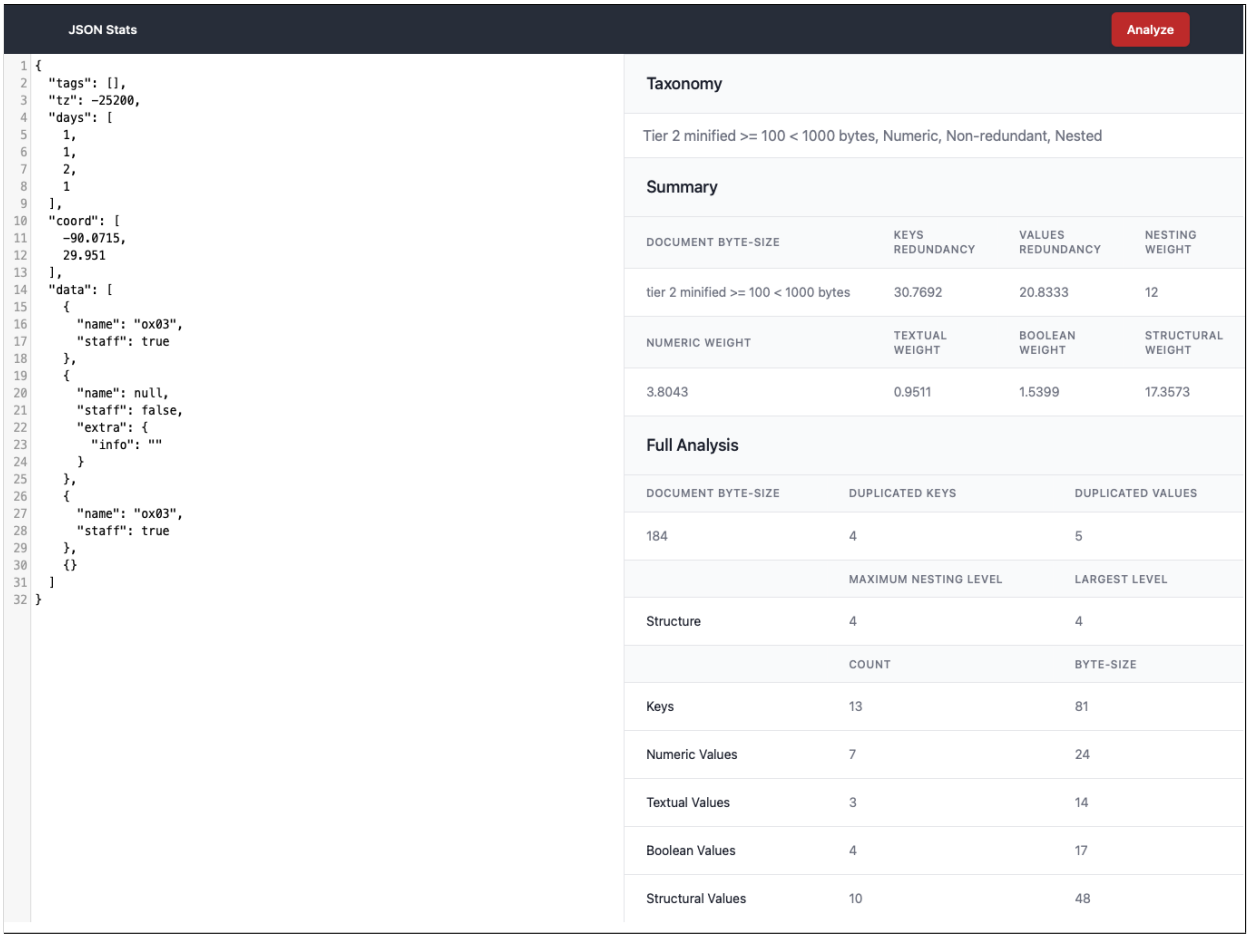

Figure 4.10: A screenshot of the online tool published at `https://www.jsonbinpack.org/stats/` analyzing the JSON document listed in Figure 4.9.

Discussion. The document under analysis in Figure 4.10 is displayed in the embedded text editor on the left side of the screen. The analysis results are present on the right side of the screen and are generated after pressing the red *Analyze* button on the top right corner. The *Taxonomy* section of the analysis table shows that the document is a *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes, numeric, non-redundant, and nested* document according to the taxonomy. The *Summary* section shows the intermediary results of the size, content type, redundancy and structural statistics introduced in subsection 4.2.1, subsection 4.2.2, subsection 4.2.3 and subsection 4.2.4, respectively. The *Full Analysis* section shows all the intermediary values used throughout every calculation.

5 | Methodology

5.1 Study of JSON-compatible Binary Serialization Specifications

5.1.1 Approach

My approach to extend the body of literature through meticulous study of JSON-compatible binary serialization specifications is based on the following methodology. While several serialization specifications have characteristics outside of the context of JSON [21], the scope of this study is limited to those characteristics that are relevant for encoding JSON documents.

1. **Identify JSON-compatible Binary Serialization Specifications.** Research and select a set of schema-driven and schema-less JSON-compatible binary serialization specifications.

2. **Craft a JSON Test Document.** Design a sufficiently complex yet succinct JSON document in an attempt to highlight the challenges of encoding JSON data. This JSON document is referred to as the *input data*.

3. **Write Schemas for the Schema-driven Serialization Specifications.** Present schemas that describe the *input data* for each of the selected schema-driven serialization specifications. The schemas are designed to produce space-efficient results given the features documented by the corresponding specifications.

4. **Serialize the JSON Test Document.** Serialize the *input data* using each of the selected binary serialization specifications.

5. **Analyze the Bit-strings Produced by Each Serialization Specification.** Study the resulting bit-strings and present annotated hexadecimal diagrams that guide the reader in understanding the inner workings of each binary serialization specification.

6. **Discuss the Characteristics of Each Serialization Specification.** For each selected binary serialization specification, discuss the characteristics, advantages and optimizations that are relevant in the context of serializing JSON [46] documents.

5.1.2 Serialization Specifications

I selected a set of general-purpose schema-driven and schema-less serialization specifications that are popular within the open-source community. Some of the selected schema-driven serialization specifications support more than one type of encoding. In these cases, I chose the most space-efficient encoding. The implementations used in this study are freely available under open-source licenses with the exception of ASN.1 [123], for which a proprietary implementation is used. The choice of JSON-compatible serialization specifications, the selected encodings and the respective implementations are documented in Table 5.1 and Table 5.2.

Table 5.1: The schema-driven binary serialization specifications, encodings and implementations discussed in this study.

Specification	Implementation	Encoding
ASN.1	OSS ASN-1Step Version 10.0.1 (proprietary)	PER Unaligned [124]
Apache Avro	Python `avro` (pip) 1.10.0	Binary Encoding [1] with no framing
Microsoft Bond	C++ library 9.0.3	Compact Binary v1 [2]
Cap'n Proto	`capnp` command-line tool 0.8.0	Packed Encoding [3]
FlatBuffers	`flatc` command-line tool 1.12.0	Binary Wire Format [4]
Protocol Buffers	Python `protobuf` (pip) 3.13.0	Binary Wire Format [5]
Apache Thrift	Python `thrift` (pip) 0.13.0	Compact Protocol [6]

[1] https://avro.apache.org/docs/current/spec.html#binary_encoding
[2] https://microsoft.github.io/bond/reference/cpp/compact__binary_8h_source.html
[3] https://capnproto.org/encoding.html#packing
[4] https://google.github.io/flatbuffers/flatbuffers_internals.html
[5] https://developers.google.com/protocol-buffers/docs/encoding
[6] https://github.com/apache/thrift/blob/master/doc/specs/thrift-compact-protocol.md

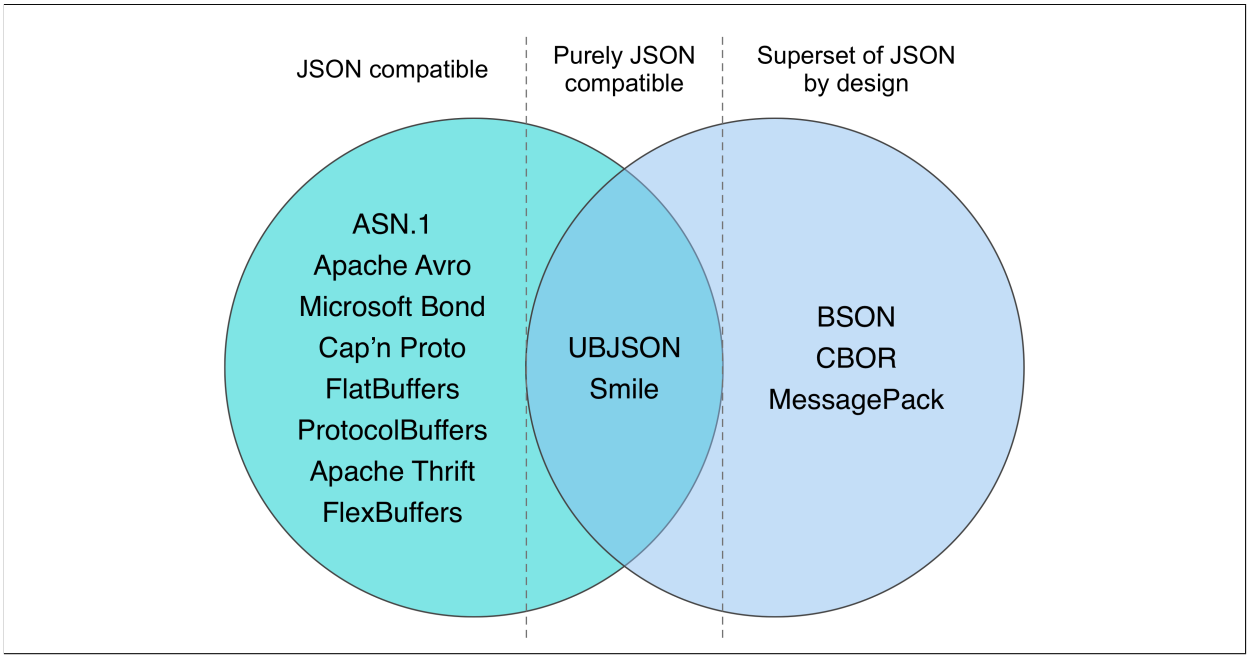

Figure 5.1: The binary serialization specifications discussed in this study divided by whether they are purely JSON compatible (center), whether they consider JSON compatibility but are a superset of JSON (right), or whether I found them to be JSON compatible but that's not one of their design goals (left).

As part of this study, I chose not to discuss serialization specifications that could not encode the *input data* JSON document from Figure 5.2 without significant changes, such as *Simple Binary Encoding* (SBE) [7] and Apple's *Binary Property List* (BPList) [8] or that could not be considered general-purpose serialization specifications, such as *Java Object Serialization* [9] and *YAS* [10]. I also chose not to discuss serialization specifications that remain unused in the industry at present, such as *PalCom Object Notation* (PON) [100] and as a consequence lack a well-documented and usable implementation, such as *SJSON* [4] and the *JSON-B*, *JSON-C* and *JSON-D* family of schema-less serialization specifications [69].

5.1.3 Input Data

I designed a test JSON [46] document that is used to showcase the challenges of serializing JSON [46] data and attempt to highlight the interesting aspects of each selected serialization specification. The JSON document I created, presented in Figure 5.2, has the following characteristics:

[7] https://github.com/real-logic/simple-binary-encoding
[8] https://opensource.apple.com/source/CF/CF-550/CFBinaryPList.c
[9] https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
[10] https://github.com/niXman/yas

Table 5.2: The schema-less binary serialization specifications, encodings and implementations discussed in this study.

Specification	Implementation
BSON	Python `bson` (pip) 0.5.10
CBOR	Python `cbor2` (pip) 5.1.2
FlexBuffers	`flatc` command-line tool 1.12.0
MessagePack	`json2msgpack` command-line tool 0.6 with `MPack` 0.9dev
Smile	Python `pysmile` (pip) 0.2
UBJSON	Python `py-ubjson` (pip) 0.16.1

```json
{
  "tags": [],
  "tz": -25200,
  "days": [ 1, 1, 2, 1 ],
  "coord": [ -90.0715, 29.9510 ],
  "data": [
    { "name": "ox03", "staff": true },
    {
      "name": null,
      "staff": false,
      "extra": { "info": "" }
    },
    { "name": "ox03", "staff": true },
    {}
  ]
}
```

Figure 5.2: The JSON test document that will be used as a basis for analyzing various binary serialization specifications.

- It contains an empty array, an empty object, and an empty string.
- It contains nested objects and homogeneous and heterogeneous arrays.
- It contains an array of scalars with and without duplicate values.
- It contains an array of composite values with and without duplicate values.
- It contains a set and an unset nullable string.
- It contains positive and negative integers and floating-point numbers.
- It contains true and false boolean values.

The *input data* is not representative of every JSON document that the reader may encounter in practice. I hope that the characteristics of the input data and output demonstrate how serialization specifications perform with respect to various JSON documents.

5.2 Benchmark of JSON-compatible Binary Serialization Specifications

5.2.1 Research Questions

Given the JSON-compatible schema-less and schema-driven binary serialization specifications studied in section 5.1, this benchmark aims to answer the following set of research questions:

- **Q1**: How do JSON-compatible schema-less binary serialization specifications compare to JSON in terms of space-efficiency?
- **Q2**: How do JSON-compatible schema-driven binary serialization specifications compare to JSON and JSON-compatible schema-less binary serialization specifications in terms of space-efficiency?

- **Q3**: How do JSON-compatible sequential binary serialization specifications compare to JSON-compatible pointer-based binary serialization specifications in terms of space-efficiency?
- **Q4**: How does compressed JSON compare to uncompressed and compressed JSON-compatible binary serialization specifications?

5.2.2 Approach

My approach to extend the body of literature through a space-efficiency benchmark of JSON-compatible binary serialization specifications is based on the following methodology:

1. **Input Data.** Select a representative set of real-world JSON [46] documents across industries according to the taxonomy defined in section 4.2.
2. **Serialization Specifications.** Drawing on research from [151], list the set of JSON-compatible schema-less and schema-driven binary serialization specifications to be benchmarked along with their respective encodings and implementations.
3. **Compression Formats.** Select a set of popular lossless data compression formats along with their respective implementations. These compression formats will be used to compress the input JSON [46] documents and bit-strings generated by the selection of binary serialization formats.
4. **Schema Definitions.** Write schema definitions for each combination of input JSON [46] document and selected schema-driven binary serialization specification.
5. **Benchmark.** Serialize each JSON [46] document using the selection of binary serialization specifications. Then, deserialize the bit-strings and compare them to the original JSON [46] documents to test that there is no accidental loss of information.
6. **Results.** Measure the byte-size of the JSON [46] documents and bit-strings generated by each binary serialization specification in uncompressed and compressed form using the selection of data compression formats.
7. **Conclusions.** Discuss the results to identify space-efficient JSON-compatible binary serialization specifications and the role of data compression in increasing space-efficiency of JSON [46] documents.

5.2.3 Input Data

Figure 5.3 categorizes the JSON [46] documents from the SchemaStore test suite introduced in section 4.1 according to the taxonomy defined in section 4.2. The SchemaStore test suite does not contain JSON [46] documents that match 9 out of the 36 categories defined in the taxonomy, particularly in the *Tier 3 Minified* $\geq$ *1000 bytes* size category which is dominated by *textual* JSON documents. I embrace these results to conclude that the missing categories do not represent instances of JSON [46] documents that are commonly encountered in practice. The missing categories are the following:

- Tier 2 Minified $\geq 100 <$ 1000 bytes Numeric Redundant Flat (SNRF)
- Tier 2 Minified $\geq 100 <$ 1000 bytes Boolean Redundant Nested (SBRN)
- Tier 2 Minified $\geq 100 <$ 1000 bytes Boolean Non-Redundant Nested (SBNN)
- Tier 3 Minified $\geq$ 1000 bytes Numeric Redundant Nested (LNRN)
- Tier 3 Minified $\geq$ 1000 bytes Numeric Non-Redundant Flat (LNNF)
- Tier 3 Minified $\geq$ 1000 bytes Numeric Non-Redundant Nested (LNNN)
- Tier 3 Minified $\geq$ 1000 bytes Boolean Redundant Nested (LBRN)
- Tier 3 Minified $\geq$ 1000 bytes Boolean Non-Redundant Flat (LBNF)

- Tier 3 Minified $\geq$ 1000 bytes Boolean Non-Redundant Nested (LBNN)

I selected a single JSON [46] document from each matching category at random. The selection of JSON documents is listed in Table 5.3 and Table 5.4. Some JSON [46] documents I selected from the SchemaStore test suite, namely *Entry Point Regulation manifest* and *.NET Core project.json*, include a top level *$schema* string property that is not considered in the benchmark. The use of this keyword is a non-standard approach to make JSON [46] documents reference their own JSON Schema [159] definitions. This keyword is not defined as part of the formats that these JSON documents represent in the SchemaStore dataset.

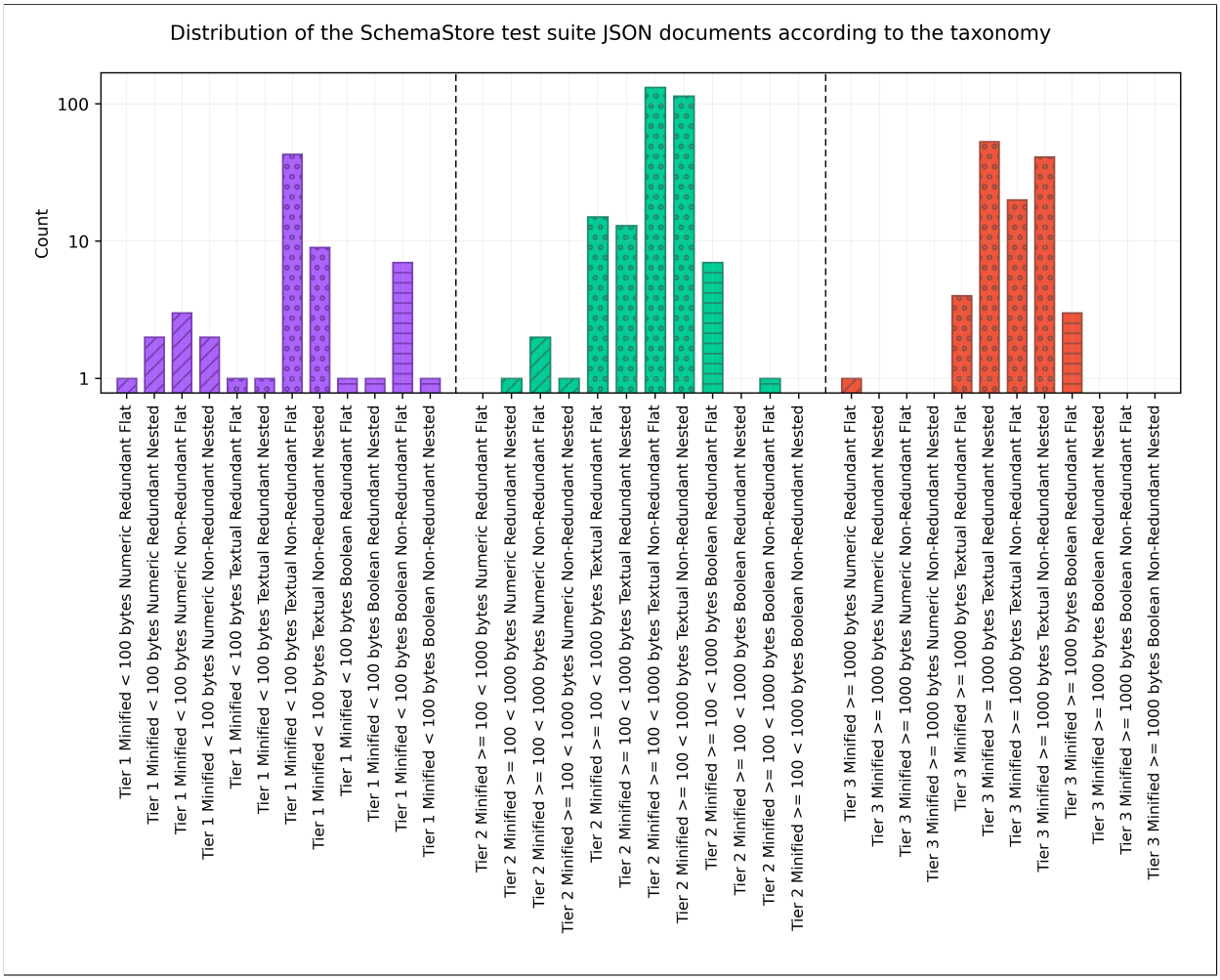

Figure 5.3: There are 480 JSON [46] documents present in the SchemaStore test suite. These are grouped according to the taxonomy defined in section 4.2 using a logarithmic y-scale.

Table 5.3: The JSON [46] documents selected from the SchemaStore test suite introduced in section 4.1 divided by industry. Each JSON document matches a different taxonomy category defined in section 4.2. The first column consists of a brief description of the JSON document. The second column contains the path and link to the test case file within the SchemaStore repository. The third column contains the taxonomy categories using the acronyms defined in Table 4.1. This table is continued in Table 5.4.

Description	Test Case Name	Category
Continuous Integration / Continuous Deliver (CI/CD)		
JSON-e templating engine sort example	jsone/sort.json	TNRF
JSON-e templating engine reverse sort example	jsone/reverse-sort.json	TNRN
CircleCI definition (blank)	circleciconfig/version-2.0.json	TNNF
CircleCI matrix definition	circleciconfig/matrix-simple.json	TNNN
SAP Cloud SDK Continuous Delivery Toolkit configuration	cloud-sdk-pipeline-config-schema/empty.json	TBRF
TravisCI notifications configuration	travis/notification-secure.json	STRF
GitHub Workflow Definition	github-workflow/919.json	STNN
Software Engineering		
Grunt.js "clean" task definition	grunt-clean-task/with-options.json	TTRF
CommitLint configuration	commitlintrc/commitlintrc-test5.json	TTRN
TSLint linter definition (extends only)	tslint/tslint-test5.json	TTNF
TSLint linter definition (multi-rule)	tslint/tslint-test25.json	TBRN
CommitLint configuration (basic)	commitlintrc/commitlintrc-test3.json	TBNF
TSLint linter definition (basic)	tslint/tslint-test19.json	TBNN
ESLint configuration document	eslintrc/WebAnalyzer.json	LNRF
NPM Package.json Linter configuration manifest	npmpackagejsonlintrc/npmpackagejsonlintrc-test.json	LTRF
.NET Core project.json	project/EF-project.json	LTRN
NPM Package.json example manifest	package/package-test.json	LTNF

Table 5.4: Continuation of Table 5.3.

Description	Test Case Name	Category
Web		
ImageOptimizer Azure Webjob configuration	`imageoptimizer/default.json`	TTNN
Entry Point Regulation manifest	`epr-manifest/official-example.json`	STRN
ECMAScript module loader definition	`esmrc/.esmrc_.json`	SBNF
Nightwatch.js Test Framework Configuration	`nightwatch/default.json`	LBRF
Geospatial		
GeoJSON example JSON document	`geojson/multi-polygon.json`	SNRN
Weather		
OpenWeatherMap API example JSON document	`openweather.current/example.json`	SNNF
OpenWeather Road Risk API example	`openweather.roadrisk/example.json`	SNNN
Publishing		
JSON Feed example document	`feed/microblog.json`	STNF
Open-Source		
GitHub FUNDING sponsorship definition (empty)	`github-funding/ebookfoundation.json`	SBRF
Recruitment		
JSON Resume	`resume/richardhendriks.json`	LTNN

5.2.4 Serialization Specifications

The selection of schema-driven and schema-less JSON-compatible binary serialization specifications is listed in Table 5.5 and Table 5.6. In comparison to my previous work [151], I use ASN-1Step 10.0.2 instead of 10.0.1, Microsoft Bond [96] 9.0.4 instead of 9.0.3, and Protocol Buffers [67] 3.15.3 instead of 3.13.0. None of these version upgrades involve changes to the encodings. Furthermore, I replaced the third-party BSON [95] Python implementation used in [151] with the Node.js official MongoDB implementation. I also replaced the Smile [121] Python implementation used in [151] with a Clojure implementation as I identified issues in the former implementation with respects to floating-point numbers. For example, encoding the floating-point number 282.55 results in 282.549988 when using `pysmile` v0.2.

Finally, both the binary and the packed encoding provided by Cap'n Proto [146] are considered. As described in [151], the packed encoding consists of a basic data compression format officially supported as a separate encoding. These encodings are separately considered to understand the impact of general-purpose data compression on the uncompressed Cap'n Proto [146] variant.

Table 5.5: The selection of schema-driven JSON-compatible binary serialization specifications based on my previous work [151].

Specification	Implementation	Encoding	License
ASN.1	OSS ASN-1Step Version 10.0.2	PER Unaligned [124]	Proprietary
Apache Avro	Python `avro` (pip) 1.10.0	Binary Encoding [11] with no framing	Apache-2.0
Microsoft Bond	C++ library 9.0.4	Compact Binary v1 [12]	MIT
Cap'n Proto	`capnp` command-line tool 0.8.0	Binary Encoding [13]	MIT
Cap'n Proto	`capnp` command-line tool 0.8.0	Packed Encoding [14]	MIT
FlatBuffers	`flatc` command-line tool 1.12.0	Binary Wire Format [15]	Apache-2.0
Protocol Buffers	Python `protobuf` (pip) 3.15.3	Binary Wire Format [16]	3-Clause BSD
Apache Thrift	Python `thrift` (pip) 0.13.0	Compact Protocol [17]	Apache-2.0

Table 5.6: The selection of schema-less JSON-compatible binary serialization specifications based on my previous work [151].

Specification	Implementation	License
BSON	Node.js `bson` (npm) 4.2.2	Apache-2.0
CBOR	Python `cbor2` (pip) 5.1.2	MIT
FlexBuffers	`flatc` command-line tool 1.12.0	Apache-2.0
MessagePack	`json2msgpack` command-line tool 0.6 with `MPack` 0.9dev	MIT
Smile	Clojure `cheshire` 5.10.0	MIT
UBJSON	Python `py-ubjson` (pip) 0.16.1	Apache-2.0

5.2.5 Fair Benchmarking

In order to produce a fair benchmark, the resulting bit-strings are ensured to be lossless encodings of the respective input JSON [46] documents. For some binary serialization specifications such as Cap'n Proto [146], providing a schema that only describes a subset of the input data will result in only such subset being serialized and the remaining of the input data being silently discarded. In other cases, a serialization specification may silently coerce an input data

[11] https://avro.apache.org/docs/current/spec.html#binary_encoding
[12] https://microsoft.github.io/bond/reference/cpp/compact__binary_8h_source.html
[13] https://capnproto.org/encoding.html#packing
[14] https://capnproto.org/encoding.html
[15] https://google.github.io/flatbuffers/flatbuffers_internals.html
[16] https://developers.google.com/protocol-buffers/docs/encoding
[17] https://github.com/apache/thrift/blob/master/doc/specs/thrift-compact-protocol.md

type to match the schema definition even at the expense of loss of information. For example, Protocol Buffers [67] may forcefully cast to IEEE 764 32-bit floating-point encoding [60] if requested by the schema even if the input real number can only be represented without loss of precision by using the IEEE 764 *64-bit* floating-point encoding [60].

The implemented benchmark program prevents such accidental mistakes by validating that for each combination of serialization specification listed in subsection 5.2.4 and input JSON document listed in Table 5.3 and Table 5.4, the produced bit-strings encode the same information as the respective input JSON document. The automated test consists in serializing the input JSON document using a given binary serialization specification, deserializing the resulting bit-string and asserting that the original JSON document is strictly equal to the deserialized JSON document.

5.2.6 Compression Formats

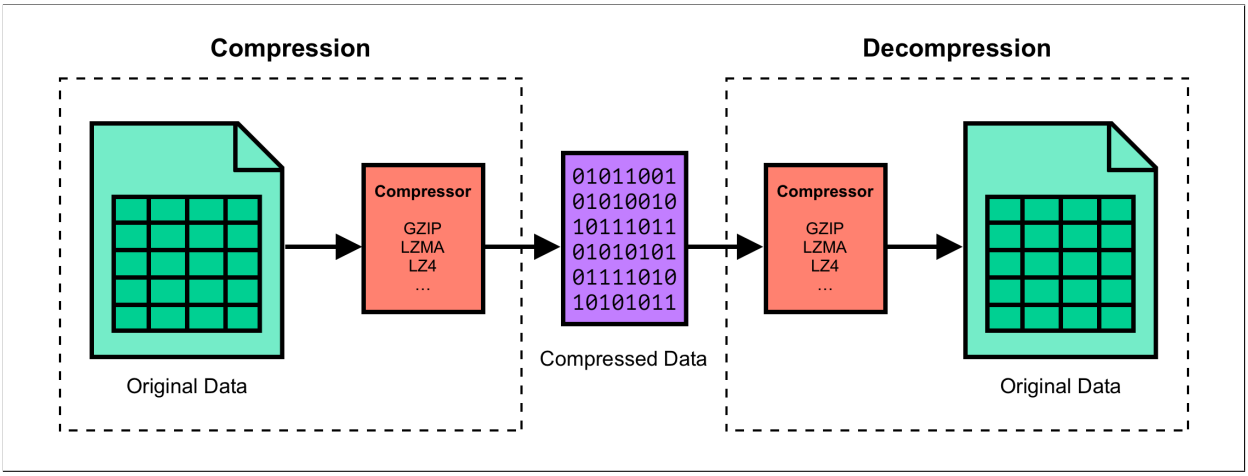

Figure 5.4: A general-purpose lossless compressor provides a *compression* and a *decompression* process. Compression consists in transforming the original data into a compressed representation of the same information. Decompression reverses the compression process to obtain the unmodified original data from its compressed representation.

I selected the following data compression formats: GZIP (GNU ZIP) [44], LZ4 [18], and Lempel-Ziv-Markov Chain Algorithm (LZMA), a set of compressors commonly used in the context of web services. These three formats are derived from the LZ77 (Lempel-Ziv) [163] dictionary-based coding scheme and are considered general-purpose lossless compressors [83]. The LZ77 [163] algorithm operates by deduplicating multiple occurrences of the same data pattern within certain distance [83] [128], a technique also discussed in more detail in [131]. According to [128], the compression ratios on textual data when using Lempel-Ziv-derived algorithm ranges between 30% and 40% in practice.

Table 5.7: A high-level view of the differences and similarities between the GZIP, LZ4 and LZMA lossless compression formats.

Compression Format	**GZIP**	**LZ4**	**LZMA**
Differences	Efficient and constant memory usage	High compression and decompression speed	Better compression on large files
Similarities	Based on LZ77 (Lempel-Ziv) [163]		

GZIP (GNU ZIP) [44] is an open-source compressor based on a mixture of the LZ77 [163] and the Huffman [74] coding schemes. GZIP was developed as part of the GNU Project [19] and released in 1992 as a replacement for the UNIX `compress` [20] program. GZIP it is the most widely-used compression format for HTTP/1.1 [56]. [54]

[18] https://lz4.github.io/lz4/
[19] https://gnu.org
[20] https://ncompress.sourceforge.io

study the problem of compressing large amounts of textual and highly-redundant data such as HTML [52] documents and use GZIP as the reference compressor due to its popularity. Their findings show that GZIP has been designed to have a small memory footprint and operate in constant space complexity [128]. These characteristics are possible given that GZIP splits the input data into small blocks of less than 1 MB and compresses each block separately. As a drawback, this approach limits GZIP ability to detect redundancy across blocks and reduces its space-efficiency on larger input data. I consider this drawback to be irrelevant for this benchmark as the largest JSON [21] document present in the SchemaStore test suite introduced in section 4.1 weights 0.5 MB as discussed in Figure 4.4.

LZ4 is an open-source compressor developed by Yann Collet [21] while working at Facebook. LZ4 is a derivative of LZ77 [163]. LZ4 focuses on improving compression and decompression speed by using a hash table data structure for storing reference addresses. The hash table provides constant $\mathcal{O}(1)$ instead of linear $\mathcal{O}(n)$ complexity for match detection [10]. LZ4 is a core part of the Zstandard data compression mechanism [37]. Zstandard is one of the eight compressors, along with GZIP [44], that are part of the IANA HTTP Content Coding Registry [22]. LZ4 is also the recommended data compression format for Cap'n Proto [146] bit-strings [23].

LZMA is an open-source compressor developed as part of the 7-Zip project [24]. LZMA offers high compression ratios as observed by [106] and [54]. LZMA is a dictionary-based compressor based on LZ77 [163] with support for dictionaries of up to 4 GB in size. As a result, LZMA can detect redundancy across large portions of the input data. In comparison to GZIP [44], [54] found LZMA to be space-efficient when taking large files as input. Applying LZMA on their 50 GB and a 440 GB collection of web pages resulted in 4.85% and 6.15% compression ratios compared to 20.35% and 18.69% compression ratios in the case of GZIP. LZMA support is implemented in the Opera web browser [25]. Official builds of the Firefox web browser do not include LZMA support. However, there exist non-official patches [26] that can be used to produce a build from source that includes LZMA support.

These data compression formats support multiple compression levels. We are interested in examining the impact of data compression in the best possible case, so I choose the highest recommended compression level supported by each format. The implementations, versions and compression levels used for this benchmark are listed in Table 5.8.

Table 5.8: The selection of lossless data compression formats.

Format	Implementation	Compression Level	License
GZIP	Apple `gzip` 321.40.3 (based on FreeBSD `gzip` 20150113)	9 [27]	2-Clause BSD
LZ4	`lz4` command-line tool v1.9.3	9 [28]	Mixed 2-Clause BSD and GPLv2
LZMA	`xz` (XZ Utils) 5.2.5 with `liblzma` 5.2.5	9 [29]	Mixed Public Domain and GNU LGPLv2.1+

5.2.7 System Specification

The implementations of the selected serialization specifications were executed on a MacBook Pro 13" Dual-Core Intel Core i5 2.9 GHz with 2 cores and 16 GB of memory (model identifier `MacBookPro12,1`) running macOS Big Sur 11.2.3, Xcode 12.4 (12D4e), clang 1200.0.32.29, GNU Make 3.81, Matplotlib 3.4.2, Awk version 20200816, Python 3.9.2, Node.js 15.11.0, and Clojure 1.10.2.796.

[21] https://github.com/Cyan4973
[22] https://www.iana.org/assignments/http-parameters/http-parameters.xhtml#content-coding
[23] https://capnproto.org/encoding.html#compression
[24] https://www.7-zip.org
[25] https://blogs.opera.com/desktop/changelog-for-31/
[26] https://wiki.mozilla.org/LZMA2_Compression
[27] https://www.unix.com/man-page/freebsd/0/gzip/
[28] https://man.archlinux.org/man/lz4.1
[29] http://manpages.org/xz

5.3 Design of a JSON-compatible Binary Serialization Specification

My approach to extend the body of literature by introducing a new space-efficient JSON-compatible binary serialization specification is based on the following methodology:

1. **Requirements**. Write a requirements specification that describes the functional and non-functional requirements of a space-efficient JSON-compatible binary serialization specification. This requirements specification is meant to capture the essential set of requirements while providing extensive space for experimentation.

2. **Characteristics**. Drawing on the research from section 5.1 and section 5.2, identify and select the combination of characteristics and constrains that tend to enable a binary serialization specification to achieve superior space-efficiency.

3. **Architecture**. Design and discuss a maintainable software architecture for a JSON-compatible binary serialization specification. The architecture shall be optimized for space-efficiency given the selected characteristics and constrains and documented using the C4 model [23] software architecture visualization notation.

4. **Implementation**. Write a proof-of-concept implementation of the JSON-compatible binary serialization specification designed in the previous steps. The choice of programming language does not affect the space-efficiency characteristics of the serialization specification. The objective is to provide a proof-of-concept implementation to evaluate the results rather than to provide a production-ready implementation.

5. **Benchmark**. Compare the space-efficiency characteristics of the new JSON-compatible serialization specification against the alternative binary serialization specifications considered in section 5.1 by extending the automated benchmark software introduced by section 5.2.

6 | A Study of JSON-compatible Binary Serialization Specifications

For a complete reference, please refer to the paper *A Survey of JSON-compatible Binary Serialization Specifications* [151]. In this particular chapter, I focus on one example, the use cases, conclusions and reproducibility sections of the above paper.

6.1 Analysis Example: ASN.1 PER Unaligned

```
00000000: 0002 9d90 0425 1002 09c0 d216 8493 74bc  .....%........t.
00000010: 6a7f 0980 d10e f9ba 5e35 3f7d 04c0 046f  j.......^5?}...o
00000020: 7830 33f8 00c0 046f 7830 3380            x03....ox03.
```

Figure 6.1: Hexadecimal output (`xxd`) of encoding Figure 5.2 input data with ASN.1 PER Unaligned (44 bytes).

History. ASN.1 [123] is a standard schema language used to serialize data structures using an extensible set of schema-driven encoding rules. ASN.1 was originally developed in 1984 as a part of the [122] standard and became a International Telecommunication Union recommendation and an ISO/IEC international standard [137] in 1988. The ASN.1 specification is publicly available [1] and there are proprietary and open source implementations of its standard encoding rules. The ASN.1 PER Unaligned encoding rules were designed to produce space-efficient bit-strings while keeping the serialization and deserialization procedures reasonably simple.

Characteristics.

- **Robustness.** ASN.1 is a mature technology that powers some of the highest-integrity communication systems in the world [2] [3] such as the Space Link Extension Services (SLE) [32] communication services for spaceflight and the LTE S1 signalling service application protocol [1]. Refer to [140] for an example of formal verification of the encoding/decoding code produced by an ASN.1 PER Unaligned compiler (Galois [4]) in the automobile industry.
- **Standardization.** In comparison to informally documented serialization specifications, ASN.1 is specified as a family of ITU Telecommunication Standardization Sector (ITU-T) recommendations and ISO/IEC international standards and has gone through extensive technical review.
- **Flexible Encoding Rules.** ASN.1 supports a wide range of standardised encodings for different use cases: BER (Basic Encoding Rules) based on tag-length-value (TLV) nested structures [125], DER (Distinguished Encoding Rules) and CER (Canonical Encoding Rules) [125] for restricted forms of BER [125], PER (Packed Encoding Rules) for space-efficiency [124], OER (Octet Encoding Rules) for runtime-efficiency [136], JER (JSON Encoding Rules) for JSON encoding [139], and XER (XML Encoding Rules) for XML encoding [138].

Layout. An ASN.1 PER Unaligned bit-string is a sequence of untagged values, sometimes nested where the ordering of the values is determined by the schema. ASN.1 PER Unaligned encodes values in as few bits as reasonably possible and does not align values to a multiple of 8 bits as the name of the encoding implies. ASN.1 PER Unaligned only encodes runtime information that cannot be inferred from the schema, such as the length of the lists or union type.

ASN.1 PER Unaligned encodes unbounded data types and bounded data types whose logical upper bound is greater than 65536 using a technique called *fragmentation* where the encoding of the value consists of one or more consecutive fragments each consisting of a length prefix followed by a series of items. The nature of each item depends on the type being encoded. For example, an item might be a character, a bit, or a logical element of a list. A value encoded using fragmentation consists of 0 or more fragments of either 16384, 32768, 49152, or 65536 items followed by a single fragment of 0 to 16383 items where each fragment is as large as possible and no fragment is larger than the preceding fragment. Refer to Table 6.1 for details on fragment length prefix encoding.

[1] https://www.itu.int/rec/T-REC-X.680/en
[2] https://www.itu.int/en/ITU-T/asn1/Pages/Application-fields-of-ASN-1.aspx
[3] https://www.oss.com/asn1/resources/standards-use-asn1.html
[4] https://galois.com

Numbers. ASN.1 PER Unaligned supports integer data types of arbitrary widths. The schema-writer may constrain the integer data type with a lower and upper bound:

- **If the integer type has no bounds or if the integer type only has an upper bound.** ASN.1 PER Unaligned encodes the value as a Big Endian Two's Complement [59] signed integer prefixed by its byte-length as a unsigned 8-bit integer.
- **If the integer type has a lower bound but not an upper bound.** ASN.1 PER Unaligned substracts the lower bound from the value and encodes the result as a variable-length Big Endian unsigned integer prefixed by its byte-length as a unsigned 8-bit integer. For example, the value -18 of an integer type whose lower bound is -20 is encoded as the unsigned integer $2 = -18 - (-20)$ prefixed with the byte-length definition `0x01`.
- **If the integer type has both a lower and an upper bound.** ASN.1 encodes the difference between the value and the lower bound using the smallest possible fixed-length Big Endian unsigned integer that can encode the difference between the upper and the lower bound. For example, an integer type constrained between the values 5 and 6 encodes the values as an unsigned 2-bit integer where 0 corresponds to 5 and 1 corresponds to 6.

In terms of real numbers, ASN.1 PER Unaligned does not support IEEE 764 floating-point numbers [60]. Instead, ASN.1 PER Unaligned encodes a real numbers as concatenation of its sign, base, scale, exponent, and mantissa where the real value equals **sign** $\times$ **mantissa** $\times$ $\mathbf{2}^{\mathbf{scale}}$ $\times$ $\mathbf{base}^{\mathbf{exponent}}$. Refer to Figure 6.2 for details on the encoding.

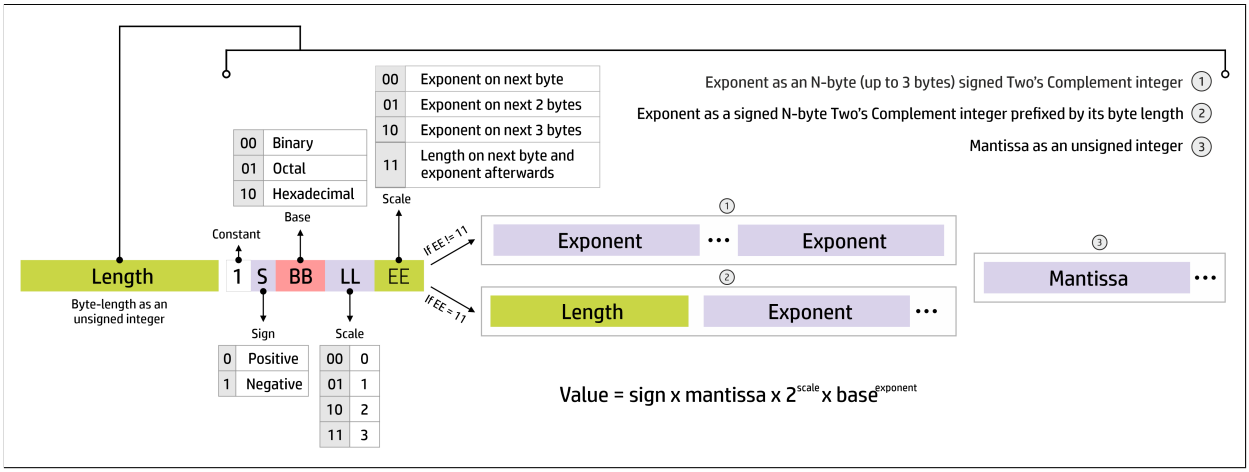

Figure 6.2: A visual representation of the **`REAL`** data type encoding inspired by [45], page 401. ASN.1 PER Unaligned encodes a real number as an 8-bit unsigned integer representing the byte-length of the real number, the sign (positive or negative) as 1-bit, the base as 2-bits (binary, octal, or hexadecimal), the scale as 2-bits (0, 1, 2, or 3), the exponent as a variable-length signed integer prefixed by its byte-length, and the mantissa as a Big Endian variable-length unsigned integer whose width is bounded by the data type length prefix.

Strings. ASN.1 supports a rich set of string types that are not *NUL*-delimited. The **`IA5String`** represents the full ASCII [33] range. The **`VisibleString`** subtype represents the subset of ASCII [33] that does not include control

Table 6.1: ASN.1 PER Unaligned fragment length prefixes depending on the number of items in the fragment as determined by the *From* and *To* ranges.

From	To	Fragment prefix
0	127	Length as an 8-bit unsigned integer
128	16383	Length as the 2-bits `10` followed by a Big Endian 14-bit unsigned integer
16384	16384	`1100 0001`
32768	32768	`1100 0010`
49152	49152	`1100 0011`
65536	65536	`1100 0100`

characters. The **`NumericString`** subtype represents digits and spaces. Finally, the **`UTF8String`** represents Unicode strings that are encoded in UTF-8 [38]. Schema-authors may subtype the supported string types to contrain the permitted alphabet. In the case of unconstrained string types and constrained string subtypes whose permitted alphabet contains more than 64 characters, each character is represented by its standard codepoint. Otherwise, ASN.1 PER Unaligned creates an ordered list of permitted characters (its alphabet) and encodes each character as an index of such list represented using the smallest possible Big Endian unsigned integer that can represent the set of permitted characters. ASN.1 PER Unaligned encodes strings using *fragmentation* when using string types in which the byte-length of the string is not always a multiple of the logical length of the string like **`UTF8String`**, when the string type has no size upper bound, or when the string type has a size upper bound which is greater than 65536. Otherwise, the string is prefixed with the string logical length as a bounded integer encoded as described in the *Numbers* section whose lower and upper bounds correspond to the size bounds of the string type.

Booleans. ASN.1 PER Unaligned encodes booleans as the bit constants 0 (False) and 1 (True).

Enumerations. ASN.1 PER Unaligned represents enumeration constants using the smallest Big Endian unsigned integer width that can encode the range of values in the enumeration.

Unions. ASN.1 supports a union operator called **`CHOICE`**. ASN.1 PER Unaligned prefixes the encoded value with the index to the choice in the union data type as the smallest-width Big Endian unsigned integer that can represent the available choices. ASN.1 also supports the concept of an *open type*. An open type is a container that holds an arbitrary value of a type known by the serializer and the deserializer applications. An open type is encoded as the encoding of the arbitrary value using *fragmentation*. ASN.1 PER Unaligned does not encode the type of the arbitrary value. Therefore, the byte length information allows a deserializer to skip the field if it does not know how to decode it.

Lists. ASN.1 PER Unaligned supports an heterogeneous list type called **`SEQUENCE`** and an homogeneous list type called **`SEQUENCE OF`**. Both sequence types can be bounded or unbounded. Unbounded sequences are encoded using *fragmentation* and empty unbounded sequences are encoded as an empty fragment. Bounded lists are encoded as the sequence of its elements with no length metadata. If an heterogeneous sequence (**`SEQUENCE`**) contains N optional values, then the sequence is prefixed by a sequence of N bits that determine whether each optional value is set.

```
TestSchema DEFINITIONS AUTOMATIC TAGS ::= BEGIN
Test ::= SEQUENCE {
    tags SEQUENCE OF UTF8String,
    tz INTEGER,
    days SEQUENCE OF INTEGER (0..6),
    coord SEQUENCE OF REAL,
    data SEQUENCE OF SEQUENCE {
        name CHOICE { alt1 UTF8String, alt2 NULL } OPTIONAL,
        staff BOOLEAN OPTIONAL,
        extra SEQUENCE { info UTF8String } OPTIONAL
    }
}
END
```

Figure 6.3: ASN.1 schema to serialize the Figure 5.2 input data.

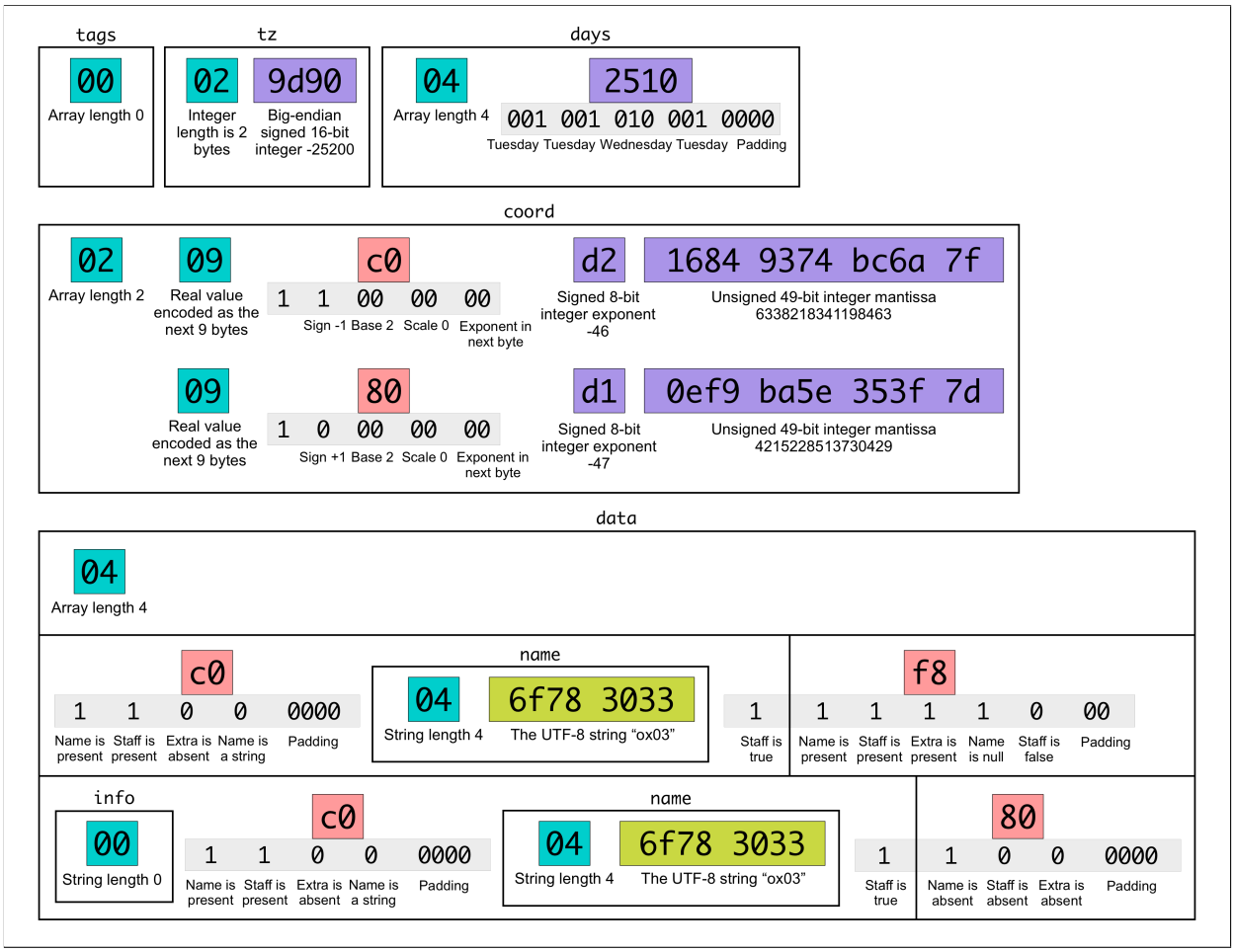

Figure 6.4: Annotated hexadecimal output of serializing the Figure 5.2 input data with ASN.1 PER Unaligned.

Table 6.2: A high-level summary of the ASN.1 PER Unaligned schema-driven serialization specification.

Website	https://www.itu.int/rec/T-REC-X.680/en
Company / Individual	International Telecommunication Union
Year	1984
Specification	ITU-T X.680-X.693 [123]
License	Implementation-dependent
Schema Language	ASN.1
Layout	Sequential and order-based
Languages	C, C++, C#, Java, Ada/Spark, Python, Erlang
Types	
Numeric	Big Endian Two's Complement [59] signed integers of user-defined length Big Endian unsigned integers of user-defined length Real numbers consisting of up to 255 bytes encoding the base, scale, exponent, and mantissa Arbitrary-length ASCII-encoded decimal numbers
String	ASCII [33], UTF-8 [38]
Composite	Choice, Enum, Set, Sequence
Scalars	Boolean, Null
Other	Octet string (byte array) Bit-string (arbitrary-length bit array) Date, Time, Date-time [80]

6.2 Use Cases

In Table 6.3, I identify a set of use-cases that binary serialization specifications tend to optimize for and the characteristics that typically enable those use-cases.

Table 6.3: Every serialization specification considered in this study supports a subset of these use-cases. The *Enabling characteristics* column describes certain characteristics that *tend* to result in a serialization specification that is a good fit for the respective use-case. The last column shows an example of a JSON-compatible binary serialization specification that has at least one of the respective enabling characteristics. However, the fact that a serialization specification has certain characteristics does not guarantee that its implementations make use of those characteristics to enable the respective use-cases, often for reasons other than technical.

Use case	**Enabling characteristics**	**Example**
Space-efficiency	The resulting bit-string embeds little metadata	ASN.1 PER Unaligned [124]
	Non-aligned data types	
Runtime-efficient deserialization	Deserialization without additional memory allocations	Cap'n Proto [146]
	Table of contents for the bit-string	
	Aligned data types	
Partial reads	Field byte-length serialized before content	FlatBuffers Binary Wire Format [143]
	Table of contents of the bit-string	
Streaming deserialization	Field byte-length serialized before content	Smile [121]
	Sequential and standalone-encoded fields	
Streaming serialization	No byte-length prefix metadata, mainly for nested structures	UBJSON [20]
	Content serialized before structure	
	Positional structural markers instead of length prefixes	
In-place updates	Field spatial locality	BSON [95]
	No byte-length field metadata	
	Positional structural markers instead of length prefixes	
Constrained devices	Simple specification and binary layout	CBOR [17]
	Small generated code and/or runtime library	
Drop-in JSON replacement	The resulting bit-string embeds all metadata	MessagePack [64]

None of the binary serialization specifications from this study support every use-case listed in Table 6.3 as some enabling characteristics tend to conflict:

- The *Space-efficiency* use-case typically involves a schema-driven serialization specification. However, JSON [21] is a schema-less serialization specification. Therefore, the *Drop-in JSON replacement* requires a schema-less serialization specification.
- The *Space-efficiency* use-case requires bit-strings to be as compact as possible. However, the *Runtime-efficient deserialization* use-case may require aligned data types and alignment may involve significant padding. For example, Cap'n Proto [146] aligns data types to 64-bit words for runtime-performance reasons and supports a simple compression scheme to mitigate the additional space overhead.
- The *Space-efficiency* use-case typically requires bit-strings to contain minimal metadata. However, the *Partial reads* use-case may require a table of contents for the bit-string, which may result in more encoded metadata. The extra overhead is amortized when encoding large amounts of data sharing the same structures. The input data JSON document from Figure 5.2 is a small data structure that consists of significant structure and

relatively little data. In the case of the Cap'n Proto [146] and FlatBuffers [143] schema-driven serialization specifications, roughly half of the bit-strings produced by serializing the input data consists of pointers and structural information that represent a table of contents.

- The *Streaming serialization* use-case may involve serializing the scalar types before the composite types, like FlexBuffers [144], given that an implementation may not know the size of a composite data type before its members are encoded. However, this approach tends to conflict with the *Streaming deserialization* use-case as an implementation would have to wait until all scalar types are received before starting to understand how they interconnect.
- The *Runtime-efficient deserialization* and *Partial reads* use-cases typically involve a pointer-based table of contents of the bit-string. As a result, implementing *In-place updates* is usually not runtime-efficient as some updates might involve adjusting pointer references in multiple parts of the bit-string as noted by [105] when using the FlatBuffers [143] serialization specification. For example, adding a new field to a FlexBuffers [144] map may involve creating a new keys vector, updating the metadata and contents of the vector data section, and adjusting most of the pointers in the bit-string.

Discussion. I could not identify a fundamental conflict involving the *Constrained devices* use-case. I believe that whether a binary serialization specification is a good fit for constrained devices tends to be a consequence of how it is implemented rather than a property of the serialization specification. For example, the official Protocol Buffers [67] implementations are typically not suitable for constrained devices as they tend to generate large amounts of code and incur significant binary size and memory allocation overheads. Kenton Varda, one of Protocol Buffers former authors, argues that the official Protocol Buffers implementations "*were designed for use in Google's servers, where binary size is mostly irrelevant, while speed and features (e.g. reflection) are valued*" [5]. However, *nanopb* [6] is a Protocol Buffers implementation targeted at 32-bit micro-controllers and other constrained devices. Refer to [14] for discussions and examples of *nanopb*.

6.3 Sequential and Pointer-based Serialization Specifications

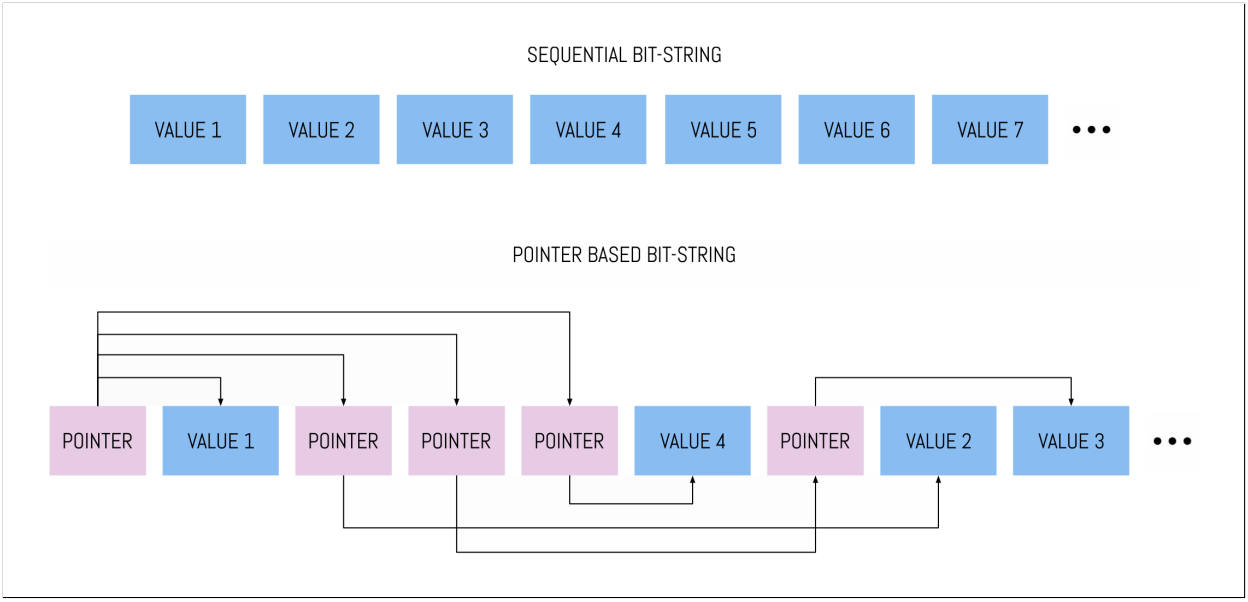

Figure 6.5: Visual representations of a sequential bit string (top) and a pointer-based bit string (bottom).

[5] https://news.ycombinator.com/item?id=25586632
[6] https://github.com/nanopb/nanopb

I found that serialization specifications can be classified into categories that are orthogonal to whether a serialization specification is schema-driven or schema-less: whether the resulting bit-string is *sequential* or *pointer-based* as shown in Figure 6.5.

Sequential. Serialization specifications are *sequential* if the bit-strings they produce are concatenations of independently-encoded data types. The majority of the serialization specifications discussed in this study are sequential. As an example, Protocol Buffers [67] is a sequential serialization specification as its bit-strings consist of a non-deterministic concatenation of fields [7] that are standalone with respect to the rest of the message.

Pointer-based. Serialization specifications are *pointer-based* if the bit-strings they produce are tree structures where each node is either a scalar type or a composite value consisting of pointers to further nodes. In comparison to sequential serialization specifications, this layout typically results in larger bit-strings that are complicated to understand. However, pointer-based serialization specifications enable efficient streaming deserialization, efficient random access reads and no additional memory allocations during deserialization which translates to better deserialization runtime performance. The pointer-based serialization specifications discussed in this study are Cap'n Proto [146], FlatBuffers [143], and FlexBuffers [144].

6.4 Reproducing this Study

The hexadecimal bit-strings discussed in this study can be recreated by the reader using the code files hosted on GitHub [8]. This GitHub repository contains a folder called *analysis* including the input data document from Figure 5.2 and the schema and code files for each binary serialization specification implementation discussed in Table 5.1 and Table 5.2. The repository includes a *Makefile* for serializing the input data document with each of the selected serialization specifications.

[7] https://developers.google.com/protocol-buffers/docs/encoding#implications
[8] https://github.com/jviotti/binary-json-survey

7 | A Benchmark of JSON-compatible Binary Serialization Specifications

For a complete reference, please refer to the paper *A Benchmark of JSON-compatible Binary Serialization Specifications* [150]. In this particular chapter, I focus on one example from each tier of the taxonomy defined in chapter 4 and the conclusions and reproducibility sections of the above paper. Example schema definitions for each presented benchmark case are shown in Figure 7.1.

7.1 Benchmark Example: Tier 1 TNN

Image Optimizer [1] is an Azure App Services WebJob [2] to compress website images used in the web development industry. In Figure 7.2, I demonstrate a **Tier 1 minified < 100 bytes textual non-redundant nested** (Tier 1 TNN from Table 4.1) JSON document that consists of an Image Optimizer configuration to perform lossy compression on images inside a particular folder.

The smallest bit-string is produced by ASN.1 PER Unaligned [124] (21 bytes), closely followed by Protocol Buffers [67] (23 bytes) and Apache Avro [61] (24 bytes). The binary serialization specifications that produced the smallest bit-strings are schema-driven and sequential [151]. Conversely, the largest bit-string is produced by BSON [95] (102 bytes), closely followed by FlatBuffers [143] (100 bytes) and Cap'n Proto Binary Encoding [146] (96 bytes). With the exception of BSON, the binary serialization specifications that produced the largest bit-strings are schema-driven and pointer-based [151]. In comparison to JSON [46] (82 bytes), binary serialization achieves a **3.9x** size reduction in the best case for this input document. However, 4 out of the 14 JSON-compatible binary serialization specifications listed in Table 5.5 and Table 5.6 result in bit-strings that are larger than JSON: Cap'n Proto Binary Encoding [146], FlatBuffers [143], BSON [95] and FlexBuffers [144]. These binary serialization specifications are either schema-less or schema-driven and pointer-based.

For this Tier 1 TNN document, the best performing schema-driven serialization specification achieves a **2.9x** size reduction compared to the best performing schema-less serialization specification: CBOR [17] and MessagePack [64] (61 bytes). As shown in Table 7.1, uncompressed schema-driven specifications provide smaller *average* and *median* bit-strings than uncompressed schema-less specifications. However, as highlighted by the *range* and *standard deviation*, uncompressed schema-driven specifications exhibit higher size reduction variability depending on the expressiveness of the schema language (i.e. how the language constructs allow you to model the data) and the size optimizations devised by its authors. With the exception of the pointer-based binary serialization specifications Cap'n Proto Binary Encoding [146] and FlatBuffers [143], the selection of schema-driven serialization specifications listed in Table 5.5 produce bit-strings that are equal to or smaller than their schema-less counterparts listed in Table 5.6. The best performing sequential serialization specification achieves a **2x** size reduction compared to the best performing pointer-based serialization specification: Cap'n Proto Packed Encoding [146] (44 bytes).

The compression formats listed in subsection 5.2.6 result in positive gains for the bit-strings produced by Cap'n Proto Binary Encoding [146] and FlatBuffers [143]. The best performing uncompressed binary serialization specification achieves a **4.1x** size reduction compared to the best performing compression format for JSON: GZIP [44] (88 bytes).

Table 7.1: A byte-size statistical analysis of the benchmark results shown in Figure 7.2 divided by schema-driven and schema-less specifications.

Category	**Schema-driven**				**Schema-less**			
	Average	**Median**	**Range**	**Std.dev**	**Average**	**Median**	**Range**	**Std.dev**
Uncompressed	45.500	28	79	31.048	76.167	72	41	14.916
GZIP (compression level 9)	55.500	48	39	14.629	88.500	86.500	27	10.720
LZ4 (compression level 9)	60	47	61	23.500	93	89	39	14.468
LZMA (compression level 9)	57.750	52	34	12.387	88.667	87.500	27	10.094

[1] https://github.com/madskristensen/ImageOptimizerWebJob
[2] https://docs.microsoft.com/en-us/azure/app-service/webjobs-create

ASN.1 - Nightwatch.js Test Framework Configuration

```
GeneratedSchema DEFINITIONS AUTOMATIC TAGS ::= BEGIN
Globals ::= SEQUENCE {
    abortOnAssertionFailure BOOLEAN,
    abortOnElementLocateError BOOLEAN,
    waitForConditionPollInterval INTEGER (0..MAX),
    waitForConditionTimeout INTEGER (0..MAX),
    throwOnMultipleElementsReturned BOOLEAN,
    suppressWarningsOnMultipleElementsReturned BOOLEAN,
    asyncHookTimeout INTEGER (0..MAX),
    unitTestsTimeout INTEGER (0..MAX),
    customReporterCallbackTimeout INTEGER (0..MAX),
    retryAssertionTimeout INTEGER (0..MAX)
}
Empty ::= SEQUENCE {}
Selenium ::= SEQUENCE {
    start_process BOOLEAN,
    cli_args Empty,
    server_path NULL,
    log_path UTF8String,
    check_process_delay INTEGER (0..MAX),
    max_status_poll_tries INTEGER (0..MAX),
    status_poll_interval INTEGER (0..MAX)
}
WebDriver ::= SEQUENCE {
    start_process BOOLEAN,
    cli_args Empty,
    server_path NULL,
    log_path UTF8String,
    check_process_delay INTEGER (0..MAX),
    max_status_poll_tries INTEGER (0..MAX),
    status_poll_interval INTEGER (0..MAX),
    process_create_timeout INTEGER (0..MAX),
    timeout_options Empty
}
DesiredCapabilities ::= SEQUENCE { browserName UTF8String }
Main ::= SEQUENCE {
    custom_commands_path NULL,
    custom_assertions_path NULL,
    page_objects_path NULL,
    globals_path NULL,
    globals Globals,
    dotenv Empty,
    persist_globals BOOLEAN,
    output_folder UTF8String,
    src_folders NULL,
    live_output BOOLEAN,
    disable_colors BOOLEAN,
    parallel_process_delay INTEGER (0..MAX),
    selenium Selenium,
    start_session BOOLEAN,
    end_session_on_fail BOOLEAN,
    test_workers BOOLEAN,
    test_runner UTF8String,
    webdriver WebDriver,
    test_settings Empty,
    launch_url UTF8String,
    silent BOOLEAN,
    output BOOLEAN,
    detailed_output BOOLEAN,
    output_timestamp BOOLEAN,
    disable_error_log BOOLEAN,
    screenshots BOOLEAN,
    log_screenshot_data BOOLEAN,
    desiredCapabilities DesiredCapabilities,
    exclude NULL,
    filter NULL,
    skipgroup UTF8String,
    sync_test_names BOOLEAN,
    skiptags UTF8String,
    use_xpath BOOLEAN,
    parallel_mode BOOLEAN,
    report_prefix UTF8String,
    unit_tests_mode BOOLEAN,
    default_reporter UTF8String
}
END
```

Apache Avro IDL - ImageOptimizer Azure Webjob Configuration

```
{
  "namespace": "schema.avro",
  "type": "record",
  "name": "benchmark",
  "fields": [
    {
      "name": "optimizations",
      "type": {
        "type": "array",
        "items": {
          "type": "record",
          "name": "optimization",
          "fields": [
            {
              "name": "includes",
              "type": {
                "type": "array",
                "items": "string"
              }
            },
            {
              "name": "excludes",
              "type": {
                "type": "array",
                "items": "string"
              }
            },
            {
              "name": "lossy",
              "type": "boolean"
            }
          ]
        }
      }
    }
  ]
}
```

Cap'n Proto IDL - GeoJSON Example Document

```
@0xef32c67826d55d2d;

struct Main {
  type @0 :Text;
  coordinates @1 :List(List(List(List(Float64))));
}
```

Figure 7.1: Examples of schema definitions written for 3 different JSON documents using 3 different interface definition languages: an ASN.1 [123] schema definition for the JSON document presented in section 7.3 (left), an Apache Avro IDL [61] schema definition for the JSON document presented in section 7.1 (top right) and a Cap'n Proto Interface Definition Language (IDL) [146] schema definition for the JSON document presented in section 7.2 (bottom right).

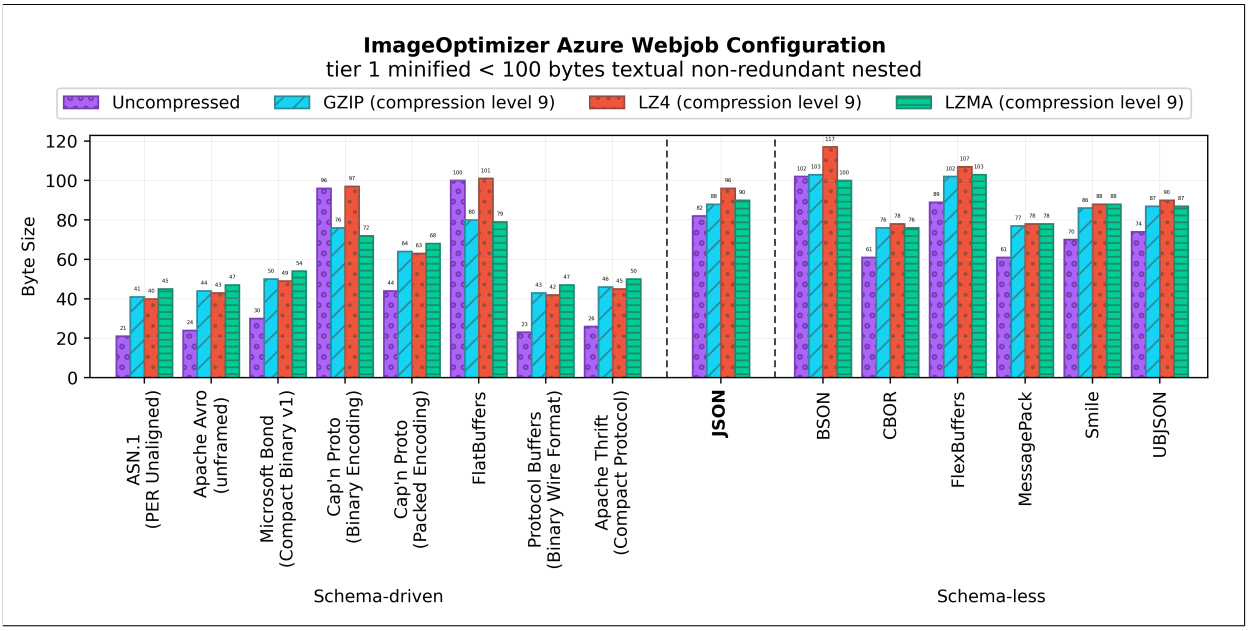

Figure 7.2: The benchmark results for the ImageOptimizer Azure Webjob Configuration test case listed in Table 5.3 and Table 5.4.

In Figure 7.3, we observe the medians for uncompressed schema-driven binary serialization specifications to be smaller in comparison to uncompressed schema-less binary serialization specifications. The range between the upper and lower whiskers and the inter-quartile range of uncompressed schema-less binary serialization specifications is smaller than the range between the upper and lower whiskers and the inter-quartile range of uncompressed schema-driven binary serialization specifications.

In terms of compression, LZ4 results in the lower median for schema-driven binary serialization specifications while GZIP results in the lower median for schema-less binary serialization specifications. However, compression is not space-efficient in terms of the median for both schema-driven and schema-less binary serialization specifications. While compression does not contribute to space-efficiency, it reduces the range between the upper and lower whiskers

Table 7.2: The benchmark raw data results and schemas for the plot in Figure 7.2.

Serialization Format	Schema	Uncompressed	GZIP	LZ4	LZMA
ASN.1 (PER Unaligned)	`schema.asn`	21	41	40	45
Apache Avro (unframed)	`schema.json`	24	44	43	47
Microsoft Bond (Compact Binary v1)	`schema.bond`	30	50	49	54
Cap'n Proto (Binary Encoding)	`schema.capnp`	96	76	97	72
Cap'n Proto (Packed Encoding)	`schema.capnp`	44	64	63	68
FlatBuffers	`schema.fbs`	100	80	101	79
Protocol Buffers (Binary Wire Format)	`schema.proto`	23	43	42	47
Apache Thrift (Compact Protocol)	`schema.thrift`	26	46	45	50
JSON	-	82	88	96	90
BSON	-	102	103	117	100
CBOR	-	61	76	78	76
FlexBuffers	-	89	102	107	103
MessagePack	-	61	77	78	78
Smile	-	70	86	88	88
UBJSON	-	74	87	90	87

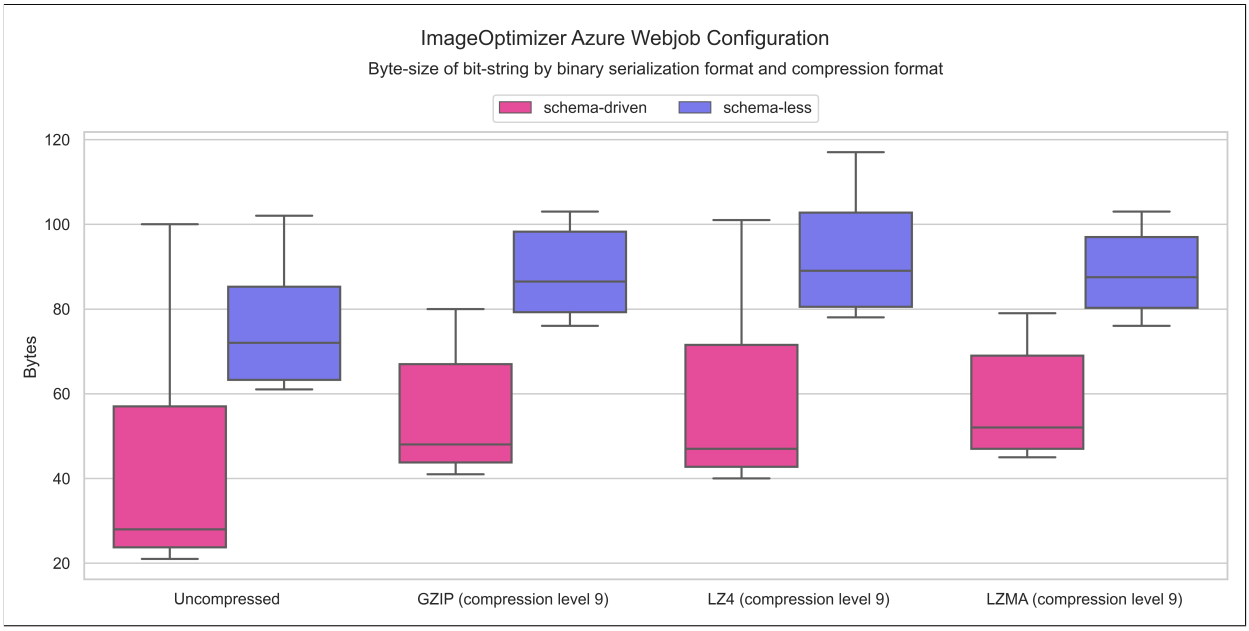

Figure 7.3: Box plot of the statistical results in Table 7.1.

and inter-quartile range for both schema-driven and schema-less binary serialization specifications. In particular, the compression format with the smaller range between the upper and lower whiskers for schema-driven binary serialization specifications is LZMA, the compression formats with the smaller inter-quartile range for schema-driven binary serialization specifications are GZIP and LZMA, the compression format with the smaller range between the upper and lower whiskers for schema-less binary serialization specifications is GZIP, and the compression format with the smaller inter-quartile range for schema-less binary serialization specifications is LZMA.

Overall, I conclude that uncompressed schema-driven binary serialization specifications are space-efficient in comparison to uncompressed schema-less binary serialization specifications and that compression does not contribute to space-efficiency in comparison to both uncompressed schema-driven and schema-less binary serialization specifications.

7.2 Benchmark Example: Tier 2 NRN

GeoJSON [27] is a standard to encode geospatial information using JSON. GeoJSON is used in industries that have geographical and geospatial use cases such as engineering, logistics and telecommunications. In Figure 7.4, I demonstrate a **Tier 2 minified $\geq$ 100 $<$ 1000 bytes numeric redundant nested** (Tier 2 NRN from Table 4.1) JSON document that defines an example polygon using the GeoJSON format.

In this exceptional case, the smallest bit-strings for this input document are produced by the schema-less sequential specifications MessagePack [64] (162 bytes) and CBOR [17] (172 bytes), followed by the schema-driven sequential specification ASN.1 PER Unaligned [124] (205 bytes). In comparison to other input documents, this input document defines poly-dimensional JSON [46] arrays, which the schema-driven binary serialization specifications from the selection do not encode in a space-efficient manner. Similarly to other cases, the largest bit-string is produced by FlatBuffers [143] (680 bytes), followed by BSON [95] (456 bytes) and Cap'n Proto Binary Encoding [146] (448 bytes). With the exception of BSON, the binary serialization specifications that produced the largest bit-strings are pointer-based [151]. In comparison to JSON [46] (190 bytes), binary serialization only achieves a **1.1x** size reduction in the best case for this input document. With the exception of the schema-less sequential MessagePack [64] and CBOR [17] binary serialization specifications, all the other JSON-compatible binary serialization specifications listed in Table 5.5 and Table 5.6 result in bit-strings that are larger than JSON.

For this Tier 2 NRN document, the smaller bit-string is produced by a schema-less specification. However, the best performing schema-less serialization specification only achieves a **1.2x** size reduction compared to the best performing schema-driven serialization specification: ASN.1 PER Unaligned [124] (205 bytes). As shown in Table 7.3,

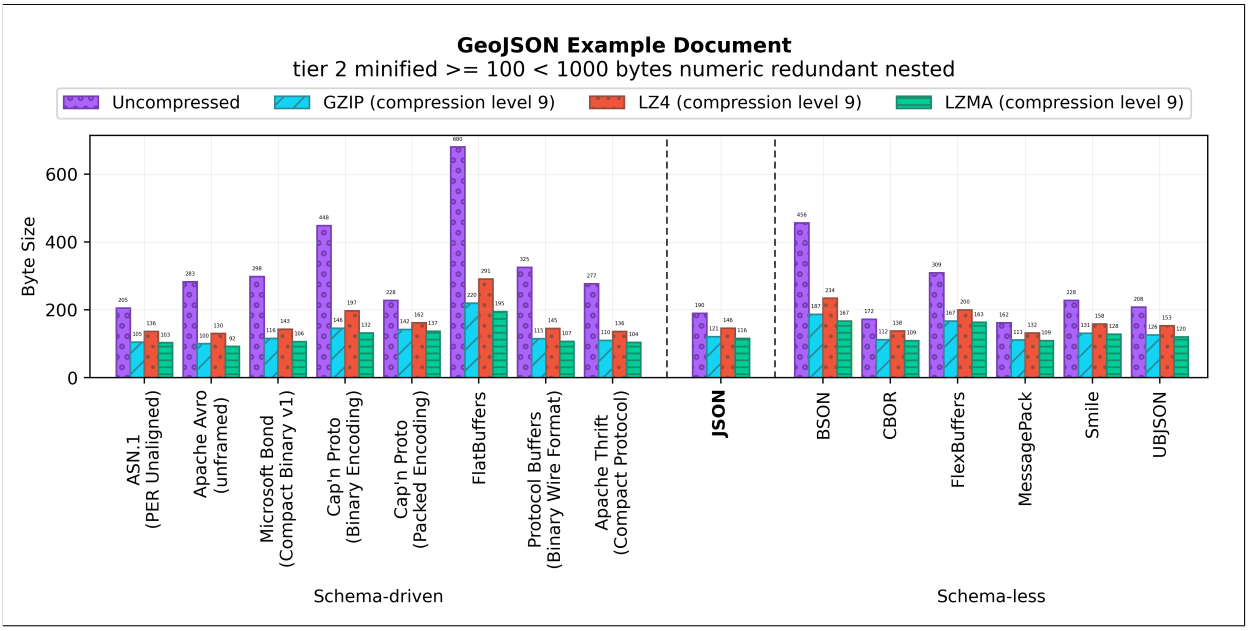

Figure 7.4: The benchmark results for the GeoJSON Example Document test case listed in Table 5.3 and Table 5.4.

uncompressed schema-less specifications provide smaller *average* and *median* bit-strings than uncompressed schema-driven specifications. Additionally, as highlighted by the *range* and *standard deviation*, uncompressed schema-driven specifications exhibit higher size reduction variability depending on the expressiveness of the schema language (i.e. how the language constructs allow you to model the data) and the size optimizations devised by its authors. The schema-less and sequential specifications CBOR [17] and MessagePack [64] produce bit-strings that are smaller to all their schema-driven counterparts listed in Table 5.5. The best performing sequential serialization specification only achieves a **1.4x** size reduction compared to the best performing pointer-based serialization specification: Cap'n Proto Packed Encoding [146] (228 bytes).

The compression formats listed in subsection 5.2.6 result in positive gains for all bit-strings. The best performing compression format for JSON, LZMA (116 bytes), achieve a **1.3x** size reduction compared to the best performing uncompressed binary serialization specification.

Table 7.3: A byte-size statistical analysis of the benchmark results shown in Figure 7.4 divided by schema-driven and schema-less specifications.

Category	**Schema-driven**				**Schema-less**			
	Average	**Median**	**Range**	**Std.dev**	**Average**	**Median**	**Range**	**Std.dev**
Uncompressed	343	290.500	475	144.554	255.833	218	294	101.480
GZIP (compression level 9)	131.750	115.500	120	36.779	139	128.500	76	28.384
LZ4 (compression level 9)	167.500	144	161	50.806	169.167	155.500	102	36.269
LZMA (compression level 9)	122	106.500	103	31.064	132.667	124	58	23.809

In Figure 7.5, contrary to other cases, we observe the medians for uncompressed schema-less binary serialization specifications to be smaller in comparison to uncompressed schema-driven binary serialization specifications. The range between the upper and lower whiskers of uncompressed schema-less binary serialization specifications is smaller than the range between the upper and lower whiskers of uncompressed schema-driven binary serialization specifications. However, the inter-quartile range of uncompressed schema-less binary serialization specifications is larger than the inter-quartile range of uncompressed schema-driven binary serialization specifications. Additionally,

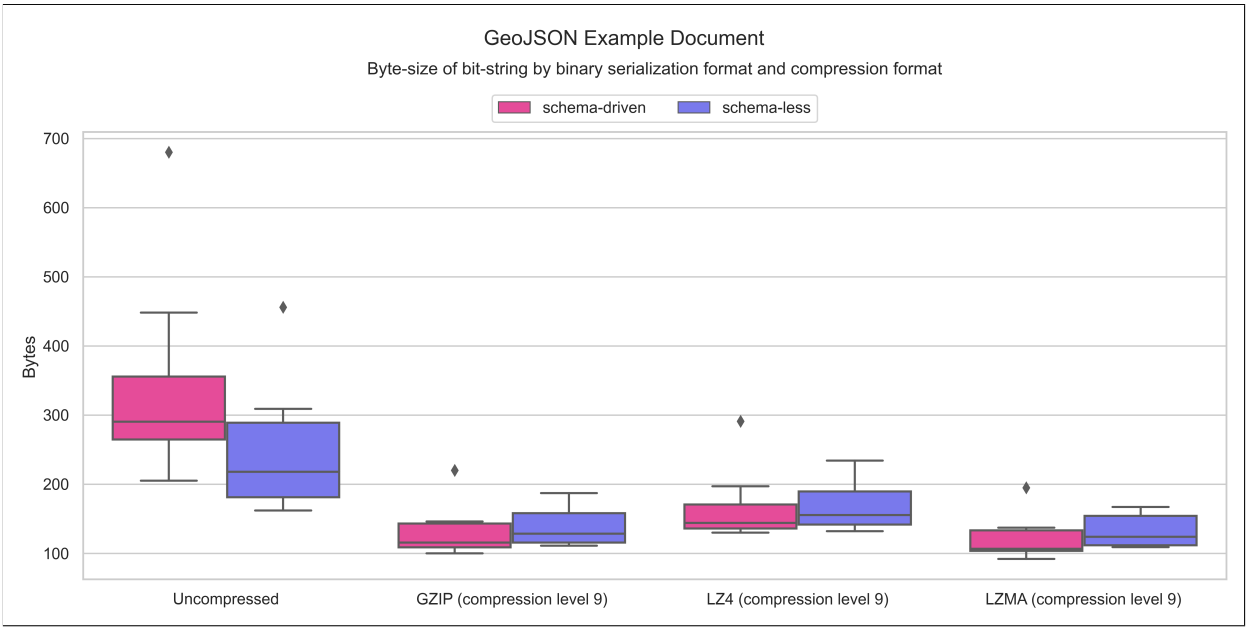

Figure 7.5: Box plot of the statistical results in Table 7.3.

their respective quartiles overlap.

In terms of compression, LZMA results in the lower median for both schema-driven and schema-less binary serialization specifications. Additionally, GZIP, LZ4 and LZMA are space-efficient in terms of the median in comparison to both uncompressed schema-driven and schema-less binary serialization specifications. However, the use of GZIP, LZ4 and LZMA for schema-driven binary serialization specifications exhibits upper outliers. Nevertheless, compression reduces the range between the upper and lower whiskers and inter-quartile range for both schema-driven and schema-less binary serialization specifications. In particular, the compression formats with the smaller range between the upper and lower whiskers and the smaller inter-quartile range for schema-driven binary serialization specifications are GZIP and LZMA, the compression format with the smaller range between the upper and lower whiskers for schema-less binary serialization specifications is LZMA, and the compression formats with the smaller inter-quartile

Table 7.4: The benchmark raw data results and schemas for the plot in Figure 7.4.

Serialization Format	**Schema**	**Uncompressed**	**GZIP**	**LZ4**	**LZMA**
ASN.1 (PER Unaligned)	`schema.asn`	205	105	136	103
Apache Avro (unframed)	`schema.json`	283	100	130	92
Microsoft Bond (Compact Binary v1)	`schema.bond`	298	116	143	106
Cap'n Proto (Binary Encoding)	`schema.capnp`	448	146	197	132
Cap'n Proto (Packed Encoding)	`schema.capnp`	228	142	162	137
FlatBuffers	`schema.fbs`	680	220	291	195
Protocol Buffers (Binary Wire Format)	`schema.proto`	325	115	145	107
Apache Thrift (Compact Protocol)	`schema.thrift`	277	110	136	104
JSON	-	190	121	146	116
BSON	-	456	187	234	167
CBOR	-	172	112	138	109
FlexBuffers	-	309	167	200	163
MessagePack	-	162	111	132	109
Smile	-	228	131	158	128
UBJSON	-	208	126	153	120

range for schema-less binary serialization specifications are GZIP and LZMA.

Overall, I conclude that uncompressed schema-less binary serialization specifications are space-efficient in comparison to uncompressed schema-driven binary serialization specifications and that all the considered compression formats are space-efficient in comparison to uncompressed schema-driven and schema-less binary serialization specifications.

7.3 Benchmark Example: Tier 3 BRF

Nightwatch.js [3] is an open-source browser automation solution used in the software testing industry. In Figure 7.6, I demonstrate a **Tier 3 minified $\geq$ 1000 bytes boolean redundant flat** (Tier 3 BRF from Table 4.1) JSON document that consists of a Nightwatch.js configuration file that defines a set of general-purpose WebDriver [130] and Selenium [4] options.

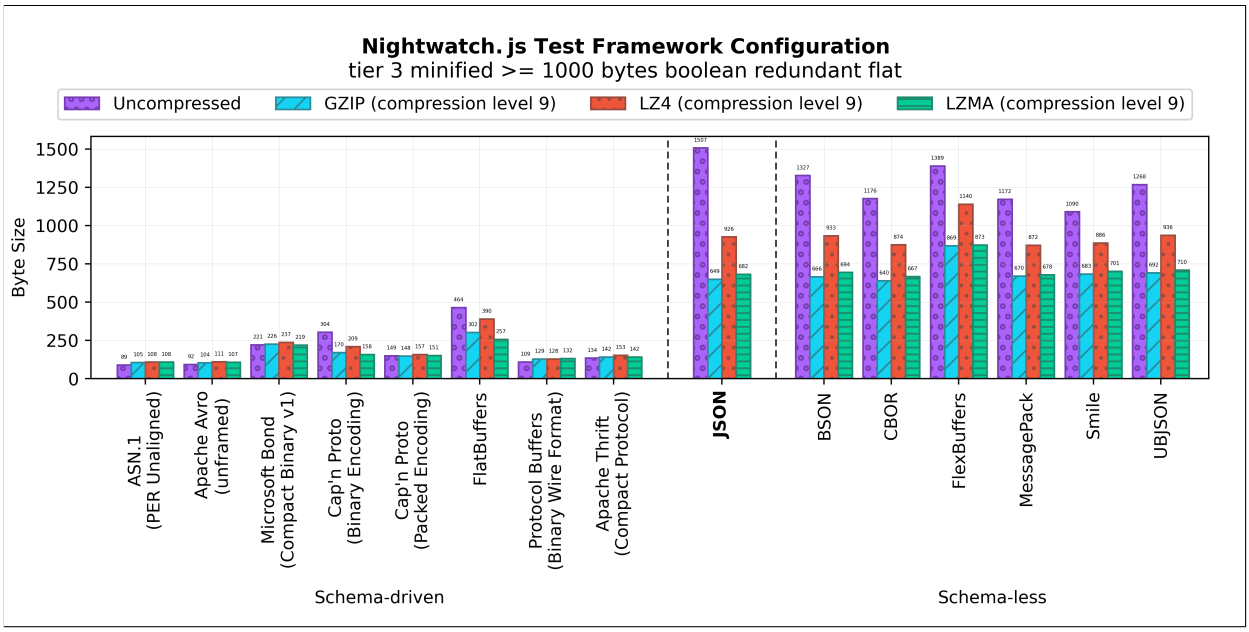

Figure 7.6: The benchmark results for the Nightwatch.js Test Framework Configuration test case listed in Table 5.3 and Table 5.4.

The smallest bit-string is produced by ASN.1 PER Unaligned [124] (89 bytes), followed by Apache Avro [61] (92 bytes) and Protocol Buffers [67] (109 bytes). The binary serialization specifications that produced the smallest bit-strings are schema-driven and sequential [151]. Conversely, the largest bit-string is produced by FlexBuffers [144] (1389 bytes), followed by BSON [95] (1327 bytes) and UBJSON [20] (1268 bytes). The binary serialization specifications that produced the largest bit-strings are schema-less and with the exception of FlexBuffers, they are also sequential [151]. In comparison to JSON [46] (1507 bytes), binary serialization achieves a **16.9x** size reduction in the best case for this input document. Similar large size reductions are observed in JSON documents whose content is dominated by *boolean* and *numeric* values. None of the 14 JSON-compatible binary serialization specifications listed in Table 5.5 and Table 5.6 result in bit-strings that are larger than JSON.

For this Tier 3 BRF document, the best performing schema-driven serialization specification achieves a **12.2x** size reduction compared to the best performing schema-less serialization specification: Smile [121] (1090 bytes). As shown in Table 7.5, uncompressed schema-driven specifications provide smaller *average* and *median* bit-strings than uncompressed schema-less specifications. However, as highlighted by the *range* and *standard deviation*, uncompressed schema-driven specifications exhibit higher size reduction variability depending on the expressiveness of the schema language (i.e. how the language constructs allow you to model the data) and the size optimizations

[3] https://nightwatchjs.org
[4] https://www.selenium.dev

devised by its authors. The entire selection of schema-driven serialization specifications listed in Table 5.5 produce bit-strings that are equal to or smaller than their schema-less counterparts listed in Table 5.6. The best performing sequential serialization specification only achieves a **1.6x** size reduction compared to the best performing pointer-based serialization specification: Cap'n Proto Packed Encoding [146] (149 bytes).

The compression formats listed in subsection 5.2.6 result in positive gains for all bit-strings except the ones produced by ASN.1 PER Unaligned [124], Apache Avro [61], Microsoft Bond [96], Protocol Buffers [67] and Apache Thrift [129]. The best performing uncompressed binary serialization specification achieves a **7.2x** size reduction compared to the best performing compression format for JSON: GZIP [44] (649 bytes).

Table 7.5: A byte-size statistical analysis of the benchmark results shown in Figure 7.6 divided by schema-driven and schema-less specifications.

Category	**Schema-driven**				**Schema-less**			
	Average	**Median**	**Range**	**Std.dev**	**Average**	**Median**	**Range**	**Std.dev**
Uncompressed	195.250	141.500	375	122.472	1237	1222	299	101.423
GZIP (compression level 9)	165.750	145	198	63.192	703.333	676.500	229	75.832
LZ4 (compression level 9)	186.625	155	282	87.853	940.167	909.500	268	93.060
LZMA (compression level 9)	159.250	146.500	150	49.487	720.500	697.500	206	69.663

Table 7.6: The benchmark raw data results and schemas for the plot in Figure 7.6.

Serialization Format	**Schema**	**Uncompressed**	**GZIP**	**LZ4**	**LZMA**
ASN.1 (PER Unaligned)	`schema.asn`	89	105	108	108
Apache Avro (unframed)	`schema.json`	92	104	111	107
Microsoft Bond (Compact Binary v1)	`schema.bond`	221	226	237	219
Cap'n Proto (Binary Encoding)	`schema.capnp`	304	170	209	158
Cap'n Proto (Packed Encoding)	`schema.capnp`	149	148	157	151
FlatBuffers	`schema.fbs`	464	302	390	257
Protocol Buffers (Binary Wire Format)	`schema.proto`	109	129	128	132
Apache Thrift (Compact Protocol)	`schema.thrift`	134	142	153	142
JSON	-	1507	649	926	682
BSON	-	1327	666	933	694
CBOR	-	1176	640	874	667
FlexBuffers	-	1389	869	1140	873
MessagePack	-	1172	670	872	678
Smile	-	1090	683	886	701
UBJSON	-	1268	692	936	710

In Figure 7.7, we observe the medians for uncompressed schema-driven binary serialization specifications to be smaller in comparison to uncompressed schema-less binary serialization specifications. The range between the upper and lower whiskers of uncompressed schema-driven binary serialization specifications is smaller than the range between the upper and lower whiskers of uncompressed schema-less binary serialization specifications. However, the inter-quartile range of both both uncompressed schema-driven and schema-less binary serialization specifications is similar.

In terms of compression, GZIP and LZMA result in the lower medians for schema-driven binary serialization specifications while GZIP results in the lower median for schema-less binary serialization specifications. Compression is not space-efficient in terms of the median in comparison to uncompressed schema-driven binary serialization specifications. However, GZIP, LZ4 and LZMA are space-efficient in terms of the median in comparison to un-

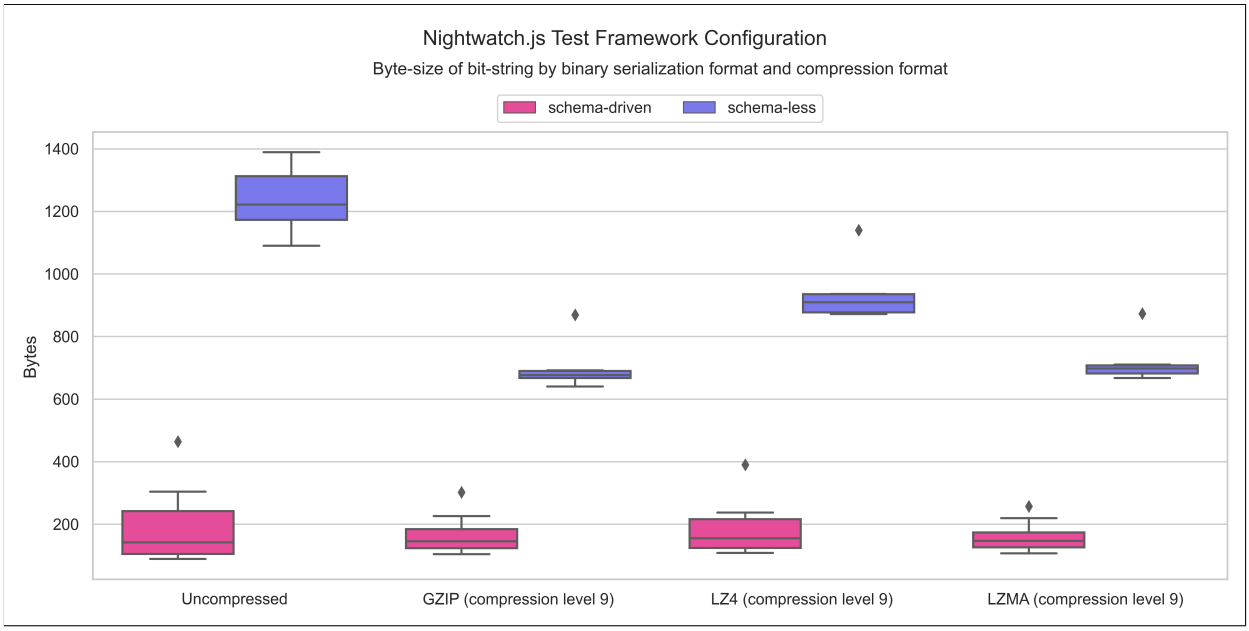

Figure 7.7: Box plot of the statistical results in Table 7.5.

compressed schema-less binary serialization specifications. Additionally, the use of GZIP, LZ4 and LZMA for both schema-driven binary serialization specifications and schema-less binary serialization specifications exhibits upper outliers. Nevertheless, compression reduces the range between the upper and lower whiskers and inter-quartile range for both schema-driven and schema-less binary serialization specifications. In particular, the compression format with the smaller inter-quartile range for schema-driven binary serialization specifications is LZMA, the compression format with the smaller range between the upper and lower whiskers for schema-less binary serialization specifications is LZMA, and the compression format with the smaller inter-quartile range for schema-less binary serialization specifications is GZIP.

Overall, I conclude that uncompressed schema-driven binary serialization specifications are space-efficient in comparison to uncompressed schema-less binary serialization specifications. Compression does not contribute to space-efficiency in comparison to schema-driven binary serialization specifications but all the considered compression formats are space-efficient in comparison to schema-less binary serialization specifications.

7.4 Conclusions

7.4.1 Q1: How do JSON-compatible schema-less binary serialization specifications compare to JSON in terms of space-efficiency?

Table 7.7 demonstrates that the median size reduction of the selection of schema-less binary serialization specifications listed in Table 5.6 is 9.1% and the average size reductions of the selection of schema-less binary serialization specifications listed in Table 5.6 is 8.2% for the selection of input data set described in Table 5.3 and Table 5.4. In comparison to JSON [46], FlexBuffers [144] and BSON [95] often result in larger bit-strings. In comparison to JSON, both CBOR [17] and MessagePack [64] are strictly superior in terms of space-efficiency. In both cases, the median and average size reductions ranged between 22.4% and 22.8% for the selection of input data. Compared to the other schema-less binary serialization specifications, MessagePack [64] tends to provide the best size reductions in the *Tier 1 Minified* $<$ *100 bytes* and *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes* categories while Smile [121] tends to provide the best size reductions for *Tier 3 Minified* $\geq$ *1000 bytes* JSON documents. As a notable positive exception shown in Figure 7.8, FlexBuffers [144] outperforms the rest of the schema-less binary serialization specifications in two cases: the *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes, textual, redundant, and flat* (Tier 2 TRF) JSON document (C) and the *Tier 3 Minified* $\geq$ *1000 bytes, textual, redundant, and flat* (Tier 3 TRF) JSON document given its automatic string deduplication features [151]. Figure 7.8 shows that CBOR [17] and MessagePack [64] tend to outperform the

other schema-less binary serialization specifications in terms of space-efficiency while producing stable results with no noticeable outliers. In comparison, BSON [95] (A and B) and FlexBuffers [144] (C and D) produce noticeable outliers at both sides of the spectrum while remaining less space-efficient than the rest of the schema-less binary serialization specifications in most cases. Like BSON [95] (B), Smile [121] produces a negative outlier (E) for the Tier 2 NRN case.

Summary. There exists schema-less binary serialization specifications that are space-efficient in comparison to JSON [46]. Based on my findings, I conclude that using MessagePack [64] on *Tier 1 Minified* $<$ *100 bytes* and *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes* JSON documents, Smile [121] on *Tier 3 Minified* $\geq$ *1000 bytes* JSON documents, and FlexBuffers [144] on JSON documents with high-redundancy of *textual* values increases space-efficiency.

Table 7.7: A summary of the size reduction results in comparison to JSON [46] of the selection of schema-less binary serialization specifications listed in Table 5.6 against the input data listed in Table 5.3 and Table 5.4. See Figure 7.8 for a visual representation of this data.

Serialization Specification	**Size Reductions in Comparison To JSON**					**Negative Cases**
	Maximum	**Minimum**	**Range**	**Median**	**Average**	
BSON	34.1%	-140.0%	174.1	-7.7%	-16.8%	21 / 27 (77.7%)
CBOR	43.2%	6.8%	36.3	22.5%	22.4%	0 / 27 (0%)
FlexBuffers	66.1%	-65.3%	131.4	-4.1%	-4.9%	16 / 27 (59.2%)
MessagePack	43.2%	6.8%	36.3	22.7%	22.8%	0 / 27 (0%)
Smile	31.8%	-20.0%	51.8	14.2%	15.5%	2 / 27 (7.4%)
UBJSON	34.1%	-9.5%	43.6	7.1%	9.9%	1 / 27 (3.7%)
Averages	**42.1%**	**-36.8%**	**78.9**	**9.1%**	**8.2%**	**24.6%**

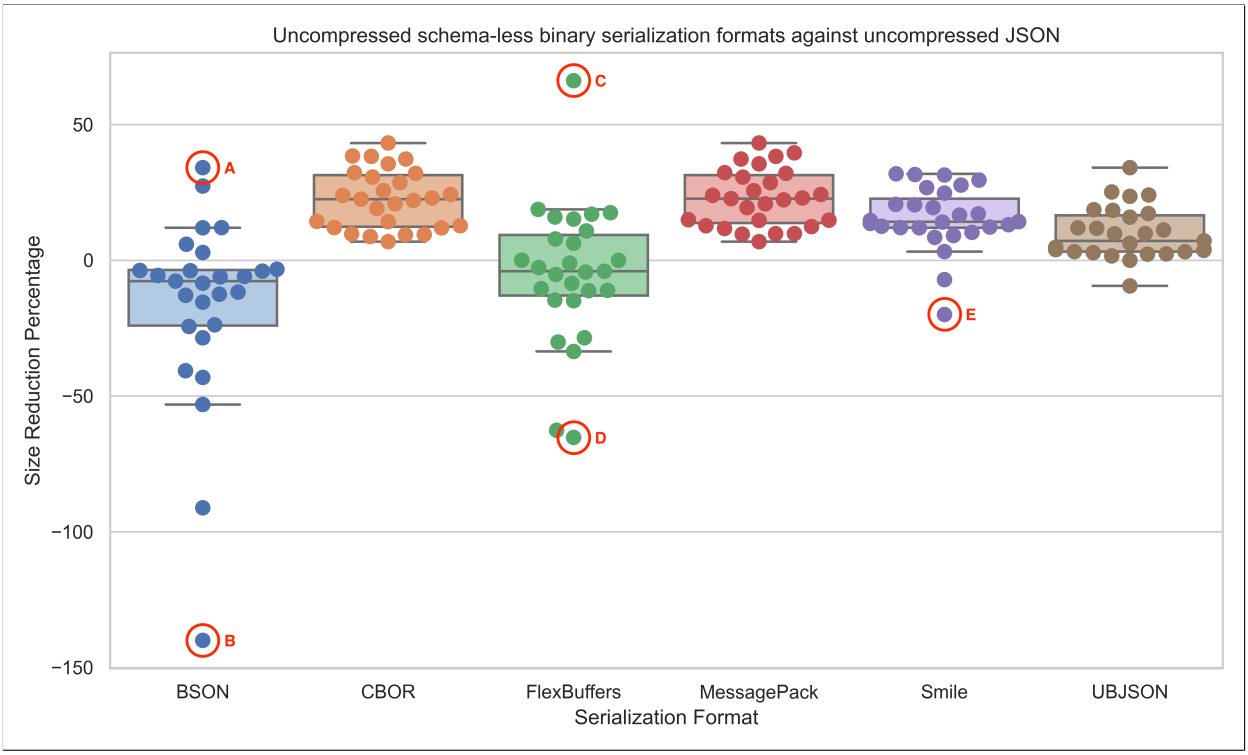

Figure 7.8: A box plot that demonstrates the size reduction (in percentages) of the selection of schema-less binary serialization specifications listed in Table 5.6 in comparison to uncompressed JSON [46] given the input data listed in Table 5.3 and Table 5.4.

7.4.2 Q2: How do JSON-compatible schema-driven binary serialization specifications compare to JSON and JSON-compatible schema-less binary serialization specifications in terms of space-efficiency?

As illustrated in Table 7.8, the median size reduction of the selection of schema-driven binary serialization specifications listed in Table 5.5 is more than five times higher that the schema-less binary serialization specification size reductions listed in Table 7.7 and the average size reduction of the selection of schema-driven binary serialization specifications listed in Table 5.5 is more than five times higher that the schema-less binary serialization specification size reductions listed in Table 7.7 for the given input data described in Table 5.3 and Table 5.4. FlatBuffers [143] and Cap'n Proto [146] (unpacked) tend to be less space-efficient than the selection of schema-less binary serialization specifications and are surpassed by the rest of the schema-driven binary serialization specifications listed in Table 5.5 for most cases. On the other side, ASN.1 PER Unaligned [124] and Apache Avro (unframed) [61] are the most space-efficient schema-driven binary serialization specifications in 23 out of the 27 cases listed in Table 5.3. Most of the schema-driven binary serialization specifications I considered are strictly superior to JSON [46] and to the schema-less binary serialization specifications listed in Table 5.6 in terms of message size with a common exception: ASN.1 PER Unaligned [124], Apache Avro (unframed) [61], Microsoft Bond (Compact Binary v1) [96], Protocol Buffers [67], Apache Thrift (Compact Protocol) [129], and Cap'n Proto [146] (packed) perform less space-efficiently than JSON [46] and the schema-less binay serialization specifications listed in Table 5.6 in the *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes, numeric, redundant, and nested* (Tier 2 NRN) GeoJSON [27] document. With the exception of ASN.1 PER Unaligned [124] and Cap'n Proto Packed Encoding [146], the selection of schema-driven binary serialization specifications result in negative outliers for the Tier 2 NRN case as shown in Figure 7.9 (A, B, C, D, E and F). Compared to the other JSON documents from the input data set, this JSON document consists of highly nested arrays and almost no object keys. Leaving that exception aside, I found that in general, the schema-driven binary serialization specifications listed Table 5.5 provide the highest space-efficiency improvements in comparison to JSON [46] on *boolean* documents and tend to provide the least space-efficient improvements on *textual* JSON documents. Figure 7.9 shows that schema-driven binary serialization specifications, in particular ASN.1 PER Unaligned [124], Apache Avro [61], Protocol Buffers [67] and Apache Thrift [129], result in high size reductions in comparison to JSON. However, every considered schema-driven binary serialization specification results in at least one negative space-efficiency exception.

Summary. The schema-driven binary serialization specifications listed in Table 5.5 tend to be more space-efficient than the schema-less binary serialization specifications listed in Table 5.6 and JSON [46] in most cases. Based on my findings, I conclude that ASN.1 PER Unaligned [124] and Apache Avro (unpacked) [61] are space-efficient in comparison to schema-less binary serialization specifications in almost all cases as they provide over 70% median size reductions and over 65% average size reductions in comparison to JSON [46].

Table 7.8: A summary of the size reduction results in comparison to JSON [46] of the selection of schema-driven binary serialization specifications listed in Table 5.5 against the input data listed in Table 5.3 and Table 5.4. See Figure 7.9 for a visual representation of this data.

Serialization Specification	**Size Reductions in Comparison To JSON**					**Negative Cases**
	Maximum	**Minimum**	**Range**	**Median**	**Average**	
ASN.1 (PER Unaligned)	98.5%	-7.9%	106.4	71.4%	65.7%	1 / 27 (3.7%)
Apache Avro (unframed)	100%	-48.9%	148.9	73.5%	65.7%	1 / 27 (3.7%)
Microsoft Bond (Compact Binary v1)	88%	-56.8%	144.8	63.4%	54%	1 / 27 (3.7%)
Cap'n Proto	81.1%	-179.1%	260.1	1.9%	-2.9%	12 / 27 (44.4%)
Cap'n Proto (packed)	90.1%	-20%	110.1	55.2%	49.6%	1 / 27 (3.7%)
FlatBuffers	72%	-257.9%	329.8	0.7%	-6.1%	13 / 27 (48.1%)
Protocol Buffers	100%	-71.1%	171.1	70.6%	59.3%	1 / 27 (3.7%)
Apache Thrift (Compact Protocol)	97.7%	-45.8%	143.5	67.6%	58.1%	1 / 27 (3.7%)
Averages	**90.9%**	**-85.9%**	**176.9**	**50.6%**	**42.9%**	**14.3%**

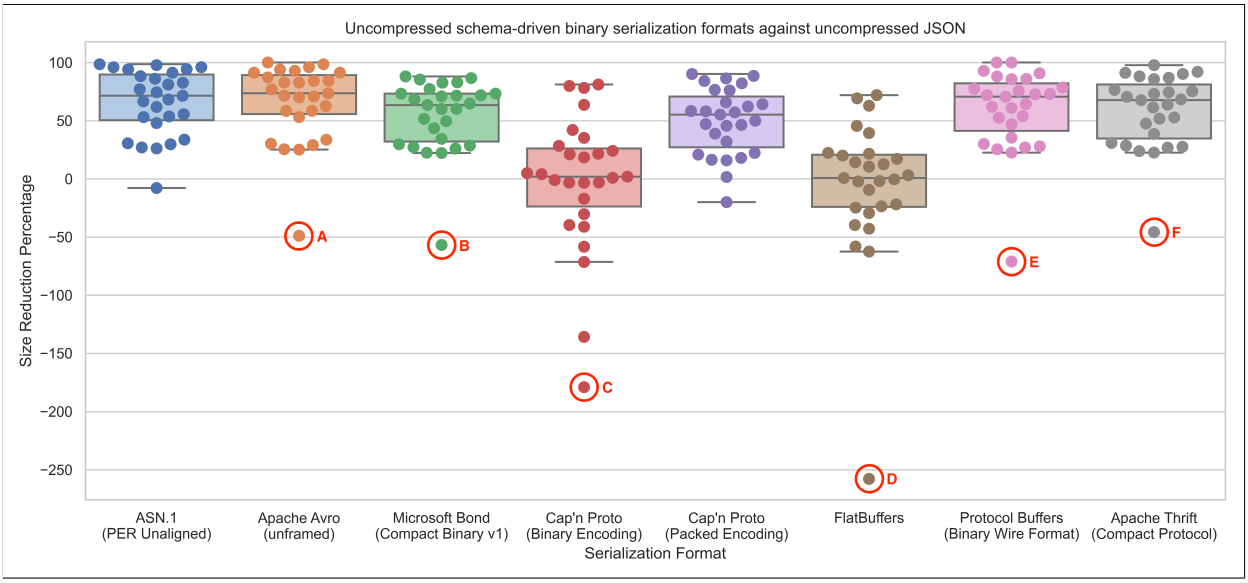

Figure 7.9: A box plot that demonstrates the size reduction (in percentages) of the selection of schema-driven binary serialization specifications listed in Table 5.5 in comparison to uncompressed JSON [46] given the input data listed in Table 5.3 and Table 5.4.

7.4.3 Q3: How do JSON-compatible sequential binary serialization specifications compare to JSON-compatible pointer-based binary serialization specifications in terms of space-efficiency?

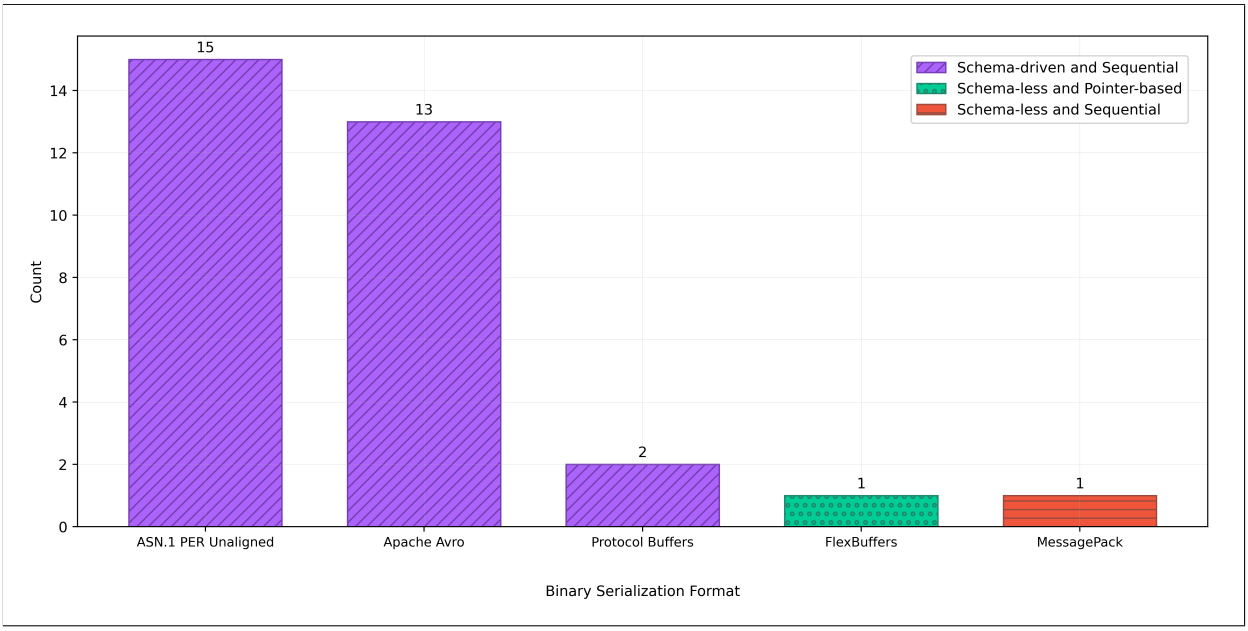

Figure 7.10: The binary serialization specifications that resulted in the highest size reductions for each JSON [46] document for the input data listed in Table 5.3 and Table 5.4, broken down by type. Schema-driven sequential binary serialization specifications, in particular ASN.1 PER Unaligned [124] and Apache Avro [61], resulted in the highest size reductions in most cases.

In terms of the schema-less binary serialization specifications listed in Table 5.6, Table 7.7 illustrates that in

comparison to JSON [46], FlexBuffers [144] results in negative median and average size reductions, a characteristic only otherwise applicable to BSON [95]. Leaving BSON aside, FlexBuffers only results in more space-efficient messages than a strict subset of the sequential schema-less binary serialization specifications in three cases: *Tier 1 TRN*, *Tier 2 TRN* and *Tier 3 TRN*. Furthemore, FlexBuffers [144] is comparatively more space-efficient that all the other schema-less binary serialization specifications listed in Table 5.6 for the *Tier 2 TRF* JSON document and the *Tier 3 TRF* JSON document. However, as explained in subsection 7.4.1, this is due to FlexBuffers automatic string deduplication feature, which is orthogonal to whether a binary serialization specification is sequential or pointer-based.

I refer to the schema-driven binary serialization specifications listed in Table 5.5. Table 7.8 illustrates that the selection of sequential schema-driven binary serialization specifications are strictly superior to FlatBuffers [143] in terms of space reductions. Similarly, Cap'n Proto [146] (unpacked) provides a more space-efficient bit-string than a single sequential schema-driven binary serialization specification, Microsoft Bond [96] (Compact Binary v1), in a single case: *Tier 2 BRF*. However, Cap'n Proto [146] (packed) results in more space-efficient messages than a strict subset of the sequential schema-driven binary serialization specifications in six cases: *Tier 1 NNF*, *Tier 1 BRF*, *Tier 2 NRN*, *Tier 2 BRF*, *Tier 3 NRF* and *Tier 3 BRF*; but never surpasses the entire set of sequential schema-driven binary serialization specifications for any JSON document from the input data set listed in Table 5.3 and Table 5.4.

Summary. Based on my findings, sequential binary serialization specifications are typically more space-efficient than pointer-based binary serialization specifications, independent of whether they are schema-less or schema-driven.

7.4.4 Q4: How does compressed JSON compares to uncompressed and compressed JSON-compatible binary serialization specifications?

7.4.4.1 Data Compression

I found that data compression tends to yield negative results on *Tier 1 Minified* < 100 *bytes* JSON documents. As an extreme, LZMA resulted in a negative 171.4% size reduction for the *Tier 1 NNF* JSON document. The entire selection of data compression formats produced negative results for all the *Tier 1 Minified* < 100 *bytes* JSON documents I considered except for the *Tier 1 BRN* JSON document, for which LZ4 produced a negative result but GZIP [44] and LZMA resulted in a 8.2% and 6.1% reduction, respectively, and the *Tier 1 TRN* JSON document, for which all data compression formats produced positive results ranging from 10.4% in the case of LZ4 to 16.7% in the case of GZIP [44]. Leaving *Tier 1 Minified* < 100 *bytes* JSON documents aside, all the data compression formats I selected offered better average and median compression ratios on *textual* JSON documents as seen in Table 7.9. Out of the selection of data compression formats, GZIP [44] performed better in terms of the average and median size reduction in all *Tier 2 Minified* $\geq 100 <$ *1000 bytes* and *Tier 3 Minified* $\geq$ *1000 bytes* categories.

Table 7.9: The average and median size reduction of using the selection of data compression formats on the *Tier 2 Minified* $\geq 100 <$ *1000 bytes* and *Tier 3 Minified* $\geq$ *1000 bytes* input JSON documents. GZIP [44] resulted in higher compression ratios for all categories.

Compression Format	**Numeric**		**Textual**		**Boolean**	
	Average	*Median*	*Average*	*Median*	*Average*	*Median*
GZIP (compression level 9)	39%	33.3%	54%	49.2%	28%	26.8%
LZ4 (compression level 9)	21%	19.5%	40%	32.7%	20%	8.7%
LZMA (compression level 9)	38%	32.8%	52%	48%	25%	21.3%

7.4.4.2 Schema-less Binary Serialization Specifications

Table 7.10 summarizes the size reductions provided by schema-less binary serialization specifications in comparison to compresed JSON [46]. Leaving BSON [95] and FlexBuffers [144] aside, schema-less binary serialization specifications typically provide space-efficient results in *Tier 1 Minified* < 100 *bytes* JSON documents, as these usually resulted in negative compression ratios. However, compressed JSON provides space-efficient results in 15 out of the 27 listed in Figure 5.3. In comparison to compressed JSON, no schema-less binary serialization provides both a positive median and average size reduction. As shown in Figure 7.11, the selection of schema-less binary serialization

specifications listed in Table 5.6, with the exception of FlexBuffers [144], result in negative outliers for the Tier 2 TRF case (A, B, C, D, E).

As summarized in Table 7.11, compressing the bit-strings produced by schema-less binary serialization specifications results in 22 out 90 instances that are space-efficient in comparison to compressed JSON on *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes* and *Tier 3 Minified* $\geq$ *1000 bytes* JSON documents but reduces the advantages that uncompressed schema-less binary serialization specifications have over compressed JSON on *Tier 1 Minified* $<$ *100 bytes* JSON documents. In comparison to compressed JSON, compressed CBOR [17] is strictly equal or superior than the rest of the compressed schema-less binary serialization specifications in all but a single case: *Tier 1 NRN*, providing the highest median (8.8%) and highest average (8.1%) size reductions. As a notable outlier shown in Figure 7.12, best-case compressed BSON [95] results in a negative size reduction of 44% in comparison to compressed JSON [46] for the Tier 2 NRN case.

Table 7.10: A summary of the size reduction results in comparison to the best case scenarios of compressed JSON [46] given the compression formats listed in Table 5.8 of the selection of schema-less binary serialization specifications listed in Table 5.6 against the input data listed in Table 5.3 and Table 5.4. See Figure 7.11 for a visual representation of this data.

Serialization Specification	**Size Reductions in Comparison To Compressed JSON**					**Negative Cases**
	Maximum	**Minimum**	**Range**	**Median**	**Average**	
BSON	50.0%	-353.9%	403.9	-40.8%	-76.9%	22 / 27 (81.4%)
CBOR	69.7%	-307.1%	376.8	7.5%	-26.8%	13 / 27 (48.1%)
FlexBuffers	45.5%	-193.5%	238.9	-48.1%	-50.8%	20 / 27 (74%)
MessagePack	69.7%	-307.1%	376.8	7.5%	-26.2%	13 / 27 (48.1%)
Smile	54.5%	-292.2%	346.8	-5%	-31.7%	14 / 27 (51.8%)
UBJSON	60.6%	-327.3%	387.9	-16.3%	-43.6%	15 / 27 (55.5%)
Averages	**58.3%**	**-296.9%**	**355.2**	**-15.9%**	**-42.7%**	**59.8%**

Table 7.11: A summary of the size reduction results of the best case scenarios of compressed schema-less binary serialization specifications listed in Table 5.6 in comparison to the best case scenarios of compressed JSON [46] given the compression formats listed in Table 5.8 and the the input data listed in Table 5.3 and Table 5.4. See Figure 7.12 for a visual representation of this data.

Serialization Specification	**Size Reductions in Comparison To Compressed JSON**					**Negative Cases**
	Maximum	**Minimum**	**Range**	**Median**	**Average**	
Compressed BSON	8%	-44%	52	-10.1%	-11%	23 / 27 (85.1%)
Compressed CBOR	24.5%	-8.7%	33.3	8.8%	8.1%	4 / 27 (14.8%)
Compressed FlexBuffers	0%	-58.9%	58.9	-24.4%	-23.8%	27 / 27 (100%)
Compressed MessagePack	24.5%	-13.7%	38.2	7.5%	5.9%	10 / 27 (37%)
Compressed Smile	13.9%	-18.4%	32.2	-1.6%	-1.6%	14 / 27 (51.8%)
Compressed UBJSON	13.6%	-16.5%	30.1	-0.7%	-1.9%	15 / 27 (55.5%)
Averages	**14.1%**	**-26.7%**	**40.8**	**-3.4%**	**-4.1%**	**57.3%**

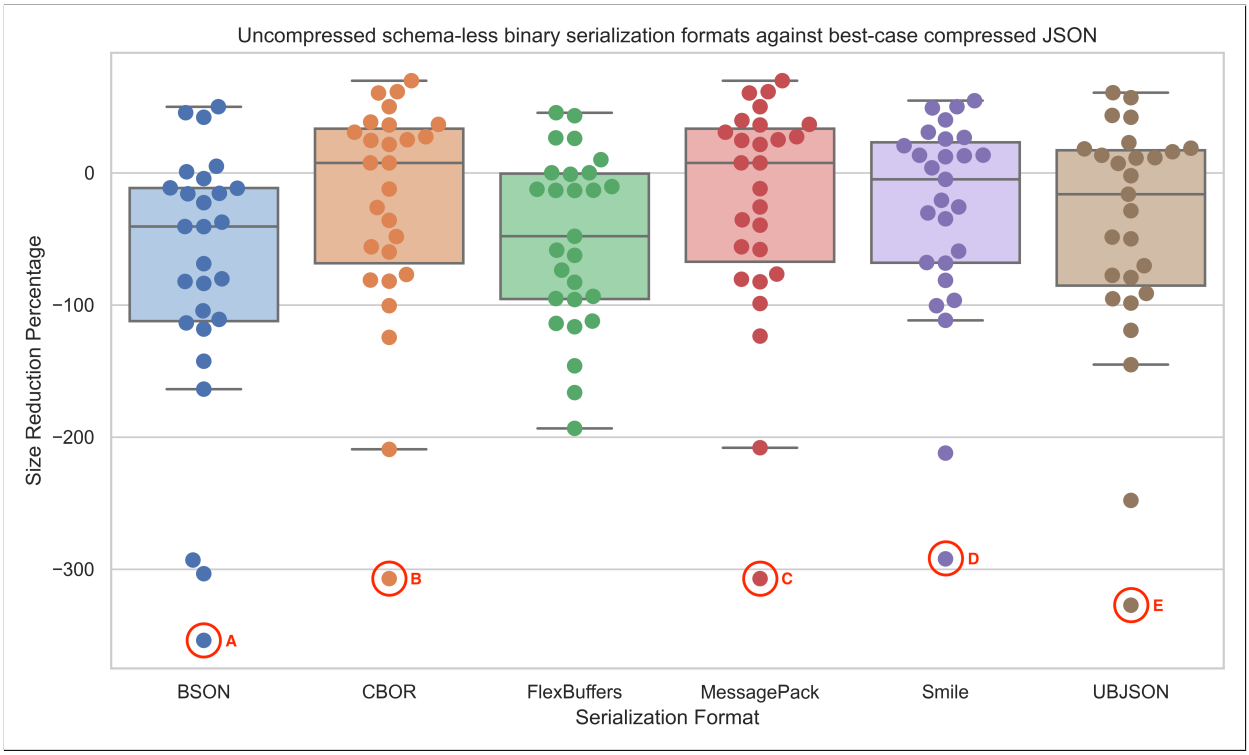

Figure 7.11: A box plot that demonstrates the size reduction (in percentages) of the selection of schema-less binary serialization specifications listed in Table 5.6 in comparison to the best-case compressed JSON [46] given the compression formats listed in Table 5.8 and the input data listed in Table 5.3 and Table 5.4.

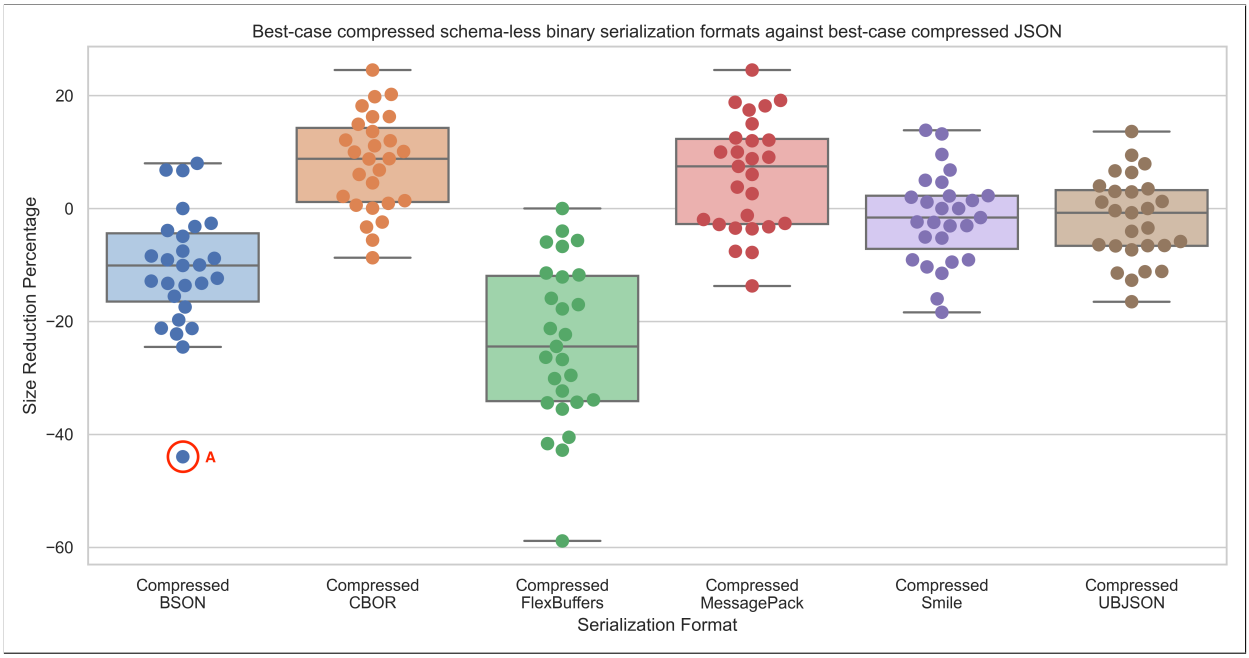

Figure 7.12: A box plot that demonstrates the size reduction (in percentages) of the selection of schema-less binary serialization specifications listed in Table 5.6 in their best-case compressed forms given the compression formats listed in Table 5.8 in comparison to the best-case compressed JSON [46] given the compression formats listed in Table 5.8 and the input data listed in Table 5.3 and Table 5.4.

7.4.4.3 Schema-driven Binary Serialization Specifications

As shown in Table 7.12, schema-driven binary serialization specifications provide positive median and average size reductions in comparison to compressed JSON [46]. However, schema-driven binary serialization specifications tend to produce negative results in comparison to compressed JSON mostly on *Tier 2 Minified* $\geq$ *100* $<$ *1000 bytes textual* (22 out of 32 cases) and *Tier 3 Minified* $\geq$ *1000 bytes textual* (25 out of 32) JSON documents. Even when taking compression into account, both ASN.1 PER Unaligned [124] and Apache Avro (unpacked) [61] continue to provide over 70% median size reductions and almost 40% average size reductions. As shown in Figure 7.13, the entire selection of schema-driven binary serialization specifications listed in Table 5.5 results in negative outliers for the Tier 2 TRF case (A, B, C, D, E, G and H) and the Tier 2 NRN case (F).

Compressing the bit-strings produced by schema-driven binary serialization specifications shows that compressed *sequential* schema-driven binary serialization specifications are strictly superior than compressed JSON [46] as shown in Table 7.13. On the higher end, both ASN.1 PER Unaligned [124] and Apache Avro [61] provide median and average size reductions of over 50% in comparison to compressed JSON, with a minimum size reduction of over 11% in the Tier 2 NRN case for which all the schema-driven binary serialization specifications previously resulted in negative size reductions in comparison to uncompressed JSON. As a notable exception shown in Figure 7.14, best-case compressed FlatBuffers [143] results in a negative size reduction of 68.1% (A) in comparison to compressed JSON [46] for the Tier 2 NRN case.

Table 7.12: A summary of the size reduction results in comparison to the best case scenarios of compressed JSON [46] given the compression formats listed in Table 5.8 of the selection of schema-diven binary serialization specifications listed in Table 5.5 against the input data listed in Table 5.3 and Table 5.4. See Figure 7.13 for a visual representation of this data.

Serialization Specification	**Size Reductions in Comparison To Compressed JSON**					**Negative Cases**
	Maximum	**Minimum**	**Range**	**Median**	**Average**	
ASN.1 (PER Unaligned)	98.5%	-222.7%	321.3	75.5%	39%	6 / 27 (22.2%)
Apache Avro (unframed)	100%	-227.3%	327.3	72.7%	39.4%	5 / 27 (18.5%)
Microsoft Bond (Compact Binary v1)	93.2%	-239%	332.1	60.2%	23.4%	6 / 27 (22.2%)
Cap'n Proto	70.1%	-315.6%	385.7	-9.1%	-45.7%	15 / 27 (55.5%)
Cap'n Proto (packed)	86.4%	-267.5%	353.9	50%	17%	8 / 27 (29.6%)
FlatBuffers	54.5%	-486.2%	540.8	-23.4%	-55.4%	17 / 27 (62.9%)
Protocol Buffers	100%	-238.3%	338.3	67%	28.4%	6 / 27 (22.2%)
Apache Thrift (Compact Protocol)	98%	-238.3%	336.3	69.3%	29%	6 / 27 (22.2%)
Averages	**87.6%**	**-279.4%**	**367**	**45.3%**	**9.4%**	**31.9%**

Summary. In comparison to compressed JSON, both compressed and uncompressed schema-less binary serialization specifications result in negative median and average size reductions. However, both compressed and uncompressed schema-driven binary serialization specifications result in positive median and average reduction. Furthermore, compressed sequential schema-driven binary serialization specifications are strictly superior to compressed JSON in all the cases from the input data.

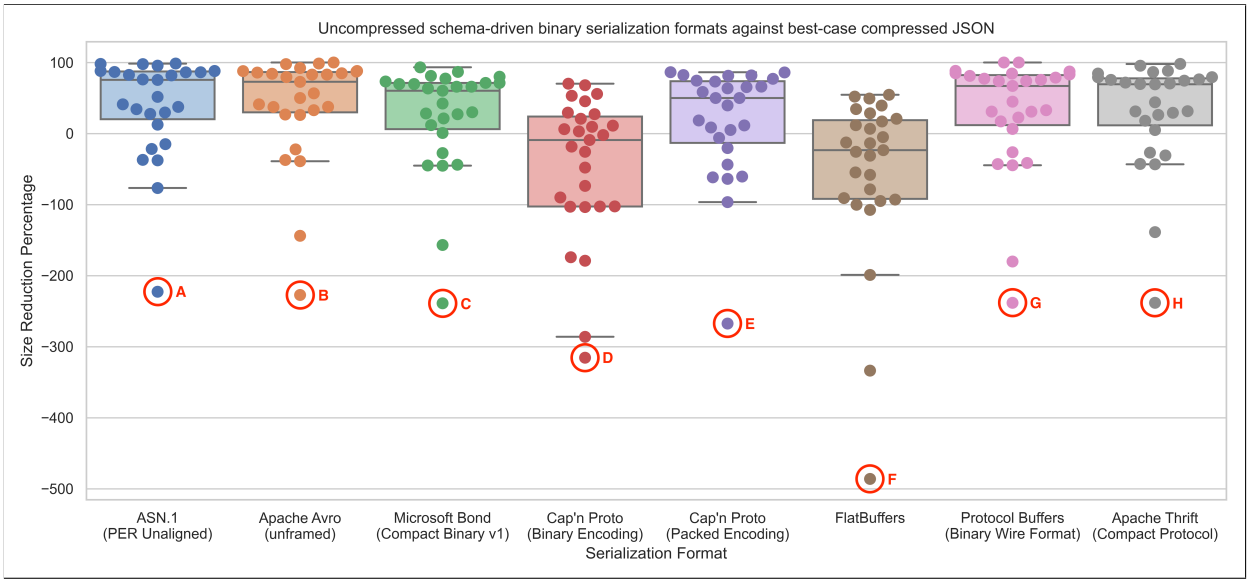

Figure 7.13: A box plot that demonstrates the size reduction (in percentages) of the selection of schema-driven binary serialization specifications listed in Table 5.5 in comparison to the best-case compressed JSON [46] given the compression formats listed in Table 5.8 and the input data listed in Table 5.3 and Table 5.4.

Table 7.13: A summary of the size reduction results of the best case scenarios of compressed schema-driven binary serialization specifications listed in Table 5.5 in comparison to the best case scenarios of compressed JSON [46] given the compression formats listed in Table 5.8 and the the input data listed in Table 5.3 and Table 5.4. See Figure 7.14 for a visual representation of this data.

Serialization Specification	**Size Reductions in Comparison To Compressed JSON**					**Negative Cases**
	Maximum	**Minimum**	**Range**	**Median**	**Average**	
Compressed ASN.1 (PER Unaligned)	83.8%	11.2%	72.6	54.5%	51.2%	0 / 27 (0%)
Compressed Apache Avro (unframed)	84%	16.7%	67.3	52.2%	52.3%	0 / 27 (0%)
Compressed Microsoft Bond (Compact Binary v1)	66.3%	8.6%	57.6	42%	40.3%	0 / 27 (0%)
Compressed Cap'n Proto	75.7%	-13.8%	89.4	22.1%	28%	3 / 27 (11.1%)
Compressed Cap'n Proto (packed)	77.2%	-18.1%	95.3	30.2%	32.7%	3 / 27 (11.1%)
Compressed FlatBuffers	60.4%	-68.1%	128.5	10.2%	12.1%	9 / 27 (33.3%)
Compressed Protocol Buffers	80.3%	7.8%	72.5	46.4%	44.6%	0 / 27 (0%)
Compressed Apache Thrift (Compact Protocol)	78.1%	10.3%	67.8	48.9%	46.2%	0 / 27 (0%)
Averages	**75.7%**	**-5.7%**	**81.4**	**38.3%**	**38.4%**	**6.9%**

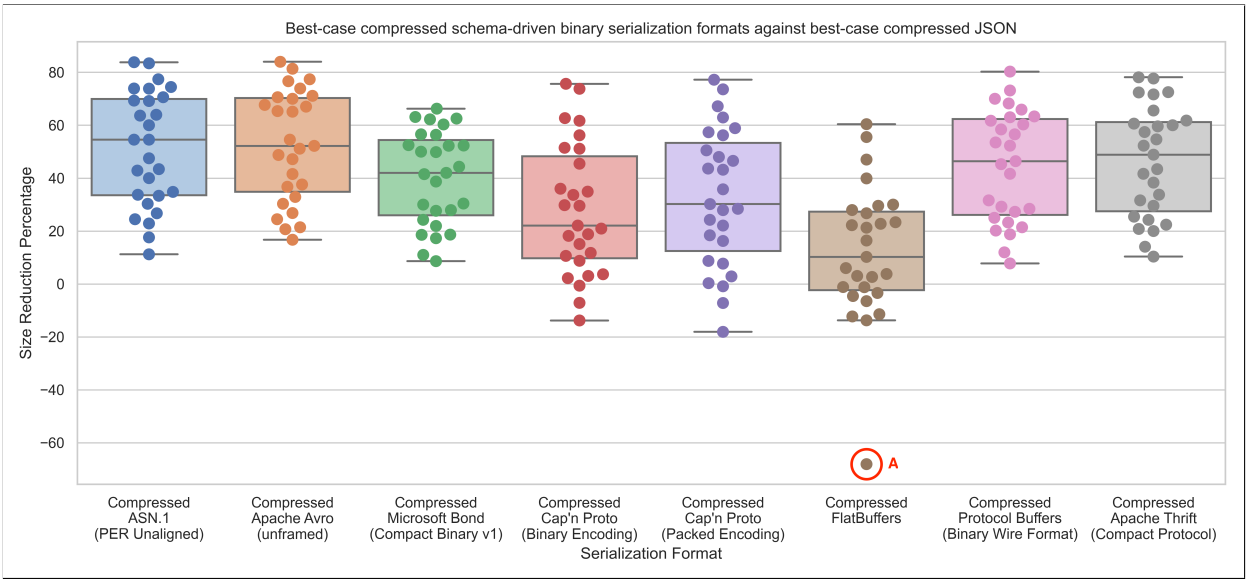

Figure 7.14: A box plot that demonstrates the size reduction (in percentages) of the selection of schema-driven binary serialization specifications listed in Table 5.5 in their best-case compressed forms given the compression formats listed in Table 5.8 in comparison to the best-case compressed JSON [46] given the compression formats listed in Table 5.8 and the input data listed in Table 5.3 and Table 5.4.

7.5 Reproducibility

To encourage reproducibility for the benchmarking datasets, I follow every reproducibility level introduced by [71]. I found that the implemented benchmark software matches the definition of Level 3, the highest-level of reproducibility, as justified in the following sections.

- **Automation.** The benchmark software, from the generation of the serialized bit-strings to the generation of the plots using Matplotlib [5], is automated through a GNU Make [6] declarative and parallelizable build definition.
- **Testing.** The POSIX shell and Python scripts distributed with the benchmark are automatically linted using the *shellcheck* [7] and *flake8* [8] open-source tools, respectively. The serialization and deserialization procedures of the benchmark are automatically tested as explained in subsection 5.2.5.
- **Supported Environments.** The benchmark software is ensured to work on the macOS (Intel processors) and GNU/Linux operating systems. I do not make any effort to support the Microsoft Windows operating system, but I expect the benchmark software to run on an *msys2* [9] or *Windows Subsystem for Linux* [10] environment with minor changes at most. The benchmark is exclusively concerned with the byte-size of the bit-strings produced by the binary serialization specifications. Therefore, the CPU, memory, and network bandwidth characteristics of the test machine do not affect the results of the benchmark. No further conditions apart from the exact software versions of the dependencies included and required by the project are necessary to replicate the results.
- **Documentation and Readability.** The `README` file [11] in the repository contains precise instructions for running the benchmark locally and generate the data files and plots. The project documentation includes a detailed list of the system dependencies that are required to succesfully execute every part of the benchmark and a detailed list of the required binary serialization specifications, implementations, versions, and encodings. The benchmark source code is compact and easy to understand and navigate due to the declarative rule definition nature of GNU Make.
- **DOI.** The version of the benchmark software described in this study is archived with a DOI [149]. The DOI includes the source code for reproducing the benchmark and the presented results.
- **Dependencies.** The benchmark software is implemented using established open-source software with the exception of the ASN-1Step [12] command-line tool, which is a proprietary implementation of ASN.1 [123] distributed by OSS Nokalva with a 30 days free trial. Every binary serialization specification implementation used in the benchmark with the exception of ASN-1Step is pinned to its specific version to ensure reproducibility. As explained in the online documentation, the benchmark software expects the ASN-1Step command-line tool version 10.0.2 to be installed and globally-accessible in the system in order to benchmark the ASN.1 PER Unaligned [124] binary serialization specification.
- **Version Control.** The benchmark repository utilises the *git* [13] version control system and its publicly hosted on GitHub [14] as recommended by [108].
- **Continuous Integration.** The GitHub repository hosting the benchmark software is setup with the GitHub Actions [15] continuous integration provided to re-run the benchmark automatically on new commits using a GNU/Linux Ubuntu 20.04 LTS cloud worker. This process prevents changes to the benchmark software from introducing regressions and new software errors. I make use of this process to validate GitHub internal and external pull requests before merging them into the trunk.

[5] https://matplotlib.org
[6] https://www.gnu.org/software/make/
[7] https://www.shellcheck.net
[8] https://flake8.pycqa.org/
[9] https://www.msys2.org
[10] https://docs.microsoft.com/en-us/windows/wsl/
[11] https://github.com/jviotti/binary-json-size-benchmark#running-locally
[12] https://www.oss.com/asn1/products/asn-1step/asn-1step.html
[13] https://git-scm.com
[14] https://github.com/jviotti/binary-json-size-benchmark
[15] https://github.com/features/actions

- **Availability.** The benchmark software and results are publicly available and governed by the *Apache License 2.0* [16] open-source software license. The results of the benchmark are also published as a website hosted at https://www.jviotti.com/binary-json-size-benchmark/ using the GitHub Pages free static hosting provider. The website provides direct links to the JSON [46] documents being encoded by the benchmark and direct links to the schema definitions used in every case. Both the JSON documents and the schema definitions are hosted in the benchmark GitHub repository to ensure their availability even if the original sources do not exist anymore.
- **Continuity.** I plan to continue extending the benchmark software in the future to test new versions of the current selection of binary serialization specifications and to include new JSON-compatible binary serialization specifications. I hope for this project to become a collaborative effort to measure the space-efficiency of every new JSON-compatible serialization specifications and I am comitted to accepting open-source contributions.

[16] https://www.apache.org/licenses/LICENSE-2.0.html

8 | Introducing JSON BinPack: A Space-efficient JSON-compatible Binary Serialization Specification

8.1 Requirements Specification

Every serialization specification is subjected to the common set of functional requirements listed in Table 8.1: S1 describes the presence of a *serialization* function, S2 describes the presence of a *deserialization* function and S3 states that the deserialization function is the *inverse* of the serialization function. While some serialization specifications enforce serialization functions to be deterministic, I consider such characteristic to be an optional feature. For example, data structures that are inherently unordered can be serialized in a non-deterministic manner regarding ordering while preserving their semantics.

Table 8.1: The common set of requirements that are implicit for any serialization specification. The fourth column proposes tests to validate whether the requirements have been met.

Identifier	Description	Type	Test
S1	A serialization specification shall describe a function to convert input data into bit-strings	Functional	The implementation exposes a *serialization* function
S2	A serialization specification shall describe a function to convert bit-strings into input data	Functional	The implementation exposes of *deserialization* function
S3	The function to convert bit-strings into input data shall reverse the process of the function to convert input data into bit-strings in all cases	Functional	The domain of the *serialization* function is often infinite. Software fuzzing is applied to validate that deserializing the result of serializing an input data structure produces the original input data structure

The JSON BinPack binary serialization specification is designed to improve network performance and reduce the operational costs of Internet-based software systems. Its architecture is further guided by the high-level non-functional requirements listed in Table 8.2. R1 captures the intention to optimize for space-efficiency. As concluded in chapter 7, the JSON-compatible binary serialization specifications that outperform the rest of the considered JSON-compatible binary serialization specifications considered in terms of this requirement are ASN.1 PER Unaligned [124] and Apache Avro [61]. R2 captures the intention to achieve strict JSON-compatibility to lower the barrier of adoption given the dominance of JSON [46] in the context of Internet-based software systems. In Appendix B, I found Apache Avro [61] to be the only schema-driven binary serialization specification that is strictly JSON-compatible, likely due to the fact that its custom schema language is also expressed using the JSON [46] data model.

Table 8.2: The set of non-functional requirements that guide the design of JSON BinPack. The fourth column proposes tests to validate whether the requirements have been met.

Identifier	Description	Type	Test
R1	JSON BinPack shall be a space-efficient serialization specification	Non-functional	JSON BinPack is space-efficient if a benchmark determines that its average and median size-reductions are lower than the ones from competing serialization specifications given a common set of representative input data
R2	JSON BinPack shall be a strictly JSON-compatible serialization specification	Non-functional	The list of valid JSON [46] documents is infinite. Software fuzzing is applied to validate that the serialization function can process every generated JSON document

8.2 Characteristics

8.2.1 Schema-driven

According to the benchmark results from chapter 7, Apache Avro [61], the most space-efficient schema-driven serialization specification, results in in 73.5% median and 65.7% average size reduction in comparison to JSON [46]. While MessagePack [64], the most space-efficient schema-less serialization specification, results in 22.7% median and 22.8% average size reduction in comparison to JSON [46].

Decision: In the interest of space-efficiency (R1 from Table 8.2), I design JSON BinPack as a *schema-driven* binary serialization specification.

8.2.2 Sequential

In section 6.3, I group serialization specifications into two categories: sequential and pointer-based. The benchmark results from chapter 7 show that in 26 out of the 27 benchmarked JSON documents, the smallest bit-string is produced by a sequential binary serialization specification.

Decision: In the interest of space-efficiency (R1 from Table 8.2), I design JSON BinPack as a *sequential* binary serialization specification.

8.2.3 Based on JSON Schema

As explained in section 3.2, JSON Schema [159] is the de-facto industry-standard schema language for JSON [46] documents. Despite this fact, JSON Schema is not adopted as the schema language of any JSON-compatible schema-driven binary serialization specification covered in chapter 6. However, there is interest in introducing JSON Schema in the context of data serialization as proven by the existance of third-party tooling that attempt to convert between JSON Schema and other schema languages used by schema-driven serialization specifications such as Protocol Buffers [67] [1] [2] [3], ASN.1 [123] [4] and Apache Avro [5]. In subsection 8.2.1, I make a case for JSON BinPack to be a schema-driven serialization specification. Therefore, the natural choice of a schema language is JSON Schema.

Decision: In the interest of JSON-compatibility (R2 from Table 8.2), I design JSON BinPack to adopt the *JSON Schema* [159] schema language.

8.2.4 Diverse Encodings

My previous work [151] reveals that serialization specifications typically encode the same data types in different manners. However, the benchmark results from chapter 7 show that no specific data type encoding is strictly superior to the rest in terms of space-efficiency. Instead, each approach may offer distinctive space-efficiency characteristics that excel in different input cases. In comparison to the serialization specifications covered in chapter 6 and chapter 7, JSON BinPack attempts to embrace the contextual nature of data type encodings and provide multiple serialization strategies to achieve space-efficiency for a wider range of input documents.

Decision: In the interest of space-efficiency (R1 from Table 8.2), I design JSON BinPack to support a diverse set of encodings and to select space-efficient serialization strategies for each input document.

[1] https://github.com/devongovett/protobuf-jsonschema
[2] https://github.com/chrusty/protoc-gen-jsonschema
[3] https://github.com/okdistribute/jsonschema-protobuf
[4] https://asn1.io/json2asn/default.aspx
[5] https://github.com/fge/json-schema-avro

8.3 Architecture

8.3.1 Overview

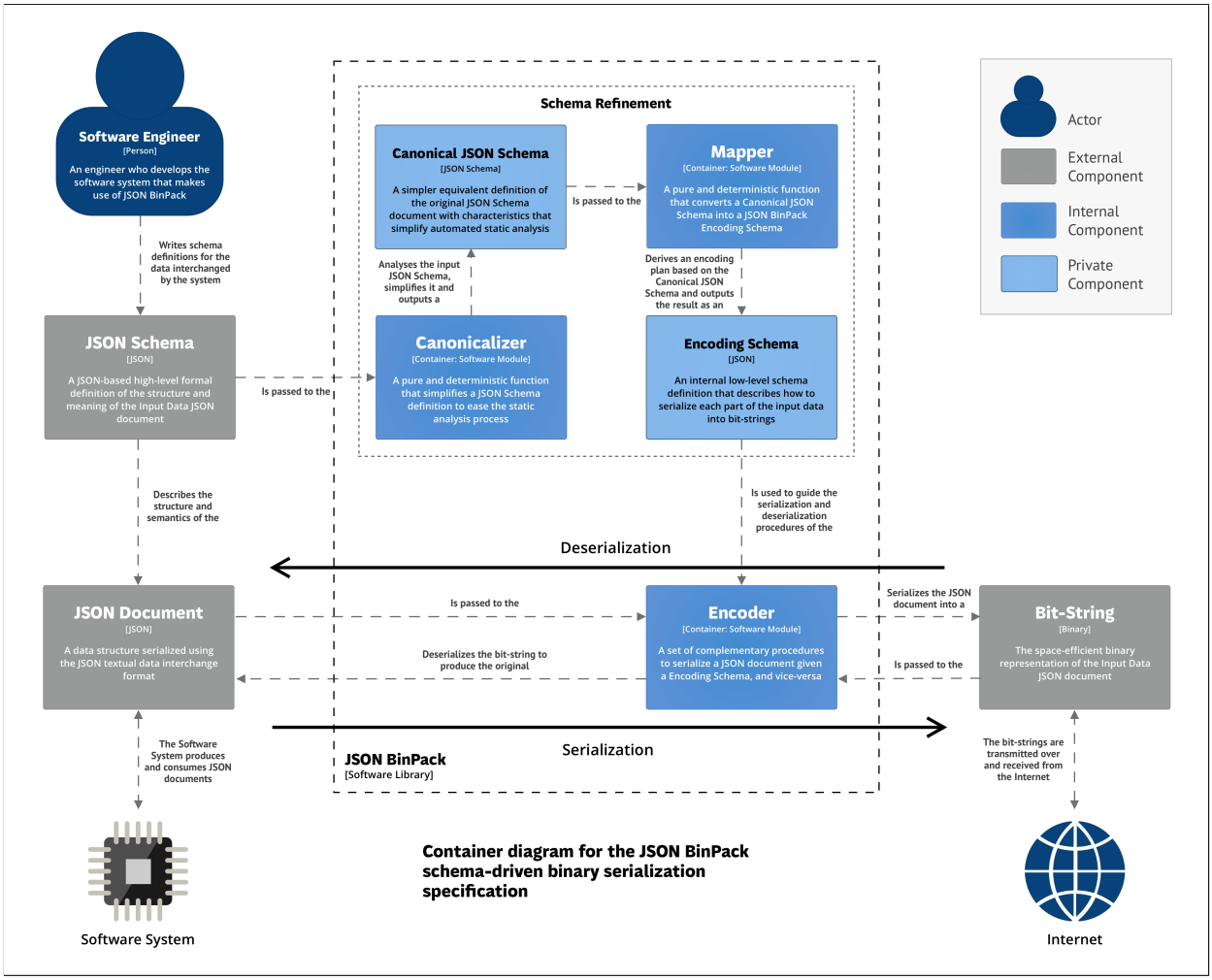

Figure 8.1: A container diagram based on the C4 software architecture visualization notation [23] for the JSON BinPack schema-driven binary serialization specification. As discussed in this section, JSON BinPack consists of three software modules: the Canonicalizer, the Encoder and the Mapper for which I provide complemental component diagrams. I present C4 diagrams for each internal component in Figure 8.3, Figure 8.4 and Figure 8.5, respectively.

8.3.1.1 Schema Refinement

For interoperatility purposes, JSON [46] and JSON Schema [159] describe data types in a high-level and implementation-independent manner. For example, JSON Schema [159] may define a JSON [46] value as a abstract homogeneous array data type and do not mandate the array to be represented in memory as a consecutive sequence of values prefixed by a header that defines the logical size of the array as some programming languages such as Java do [148]. Therefore, the core goal of JSON BinPack is to *unambiguously and bi-directionally map the high-level data types provided by JSON [46] and JSON Schema [159] to low-level space-efficient counterparts*. In order to provide these mappings, JSON BinPack statically analyses the input JSON Schema [159] definition to derive encoding and decoding rules. We may think of this process as a *schema refinement* phase that takes a high-level schema definition that describes an abstract data type and augments it to produce a new schema definition, referred to as the *encoding schema*, that more

precisely describes how to represent such abstract data type as a bit-string. The presence of an encoding schema definition for the input data is not enough by itself to perform serialization and deserialization. To complement schema refinement, I introduce an *encoder* phase that makes use of an encoding schema definition to guide the serialization and deserialization processes.

8.3.1.2 Build Time Schema Refinement

In terms of the phases introduced in this section, the schema refinement phase is concerned with the high-level schema definition while the encoder is concerned with the input data instance and the encoding schema definition. Schema-driven serialization specifications typically exploit this separation of concerns to optimize for runtime-efficiency by performing the phases that do not involve the input data at build time. Given the complexity of JSON Schema [159] as discussed in section 3.2, the JSON BinPack schema refinement phase is computationally-expensive yet crucial to achieve space-efficiency. JSON BinPack exploits this separation of concerns by allocating unbounded build time resources to the schema refinement phase and its static analysis process to derive space-efficient encoding rules. The schema-refinement phase is performed once per schema rather than once per input data instance, reducing the amount of computation required to serialize and deserialize multiple input data instances described by the same schema definition to a minimum.

8.3.1.3 Encoding Schema

Schema-driven serialization specifications such as Microsoft Bond [96], Cap'n Proto [146], Protocol Buffers [67] and Apache Thrift [129] analyze schema definitions at build time to generate source code that performs serialization and deserialization for any matching instance. In comparison to generic serialization and deserialization procedures parameterized with schema definitions, tailor-made procedures created through code-generation typically result in runtime-efficient serialization and deserialization at the expense of having to maintain code generators for multiple programming languages. For simplicity, JSON BinPack refrains from generating source code and instead generates an internal declarative low-level schema, the *encoding schema*, that is programming language agnostic. Refer to Figure 8.2 for an example encoding schema.

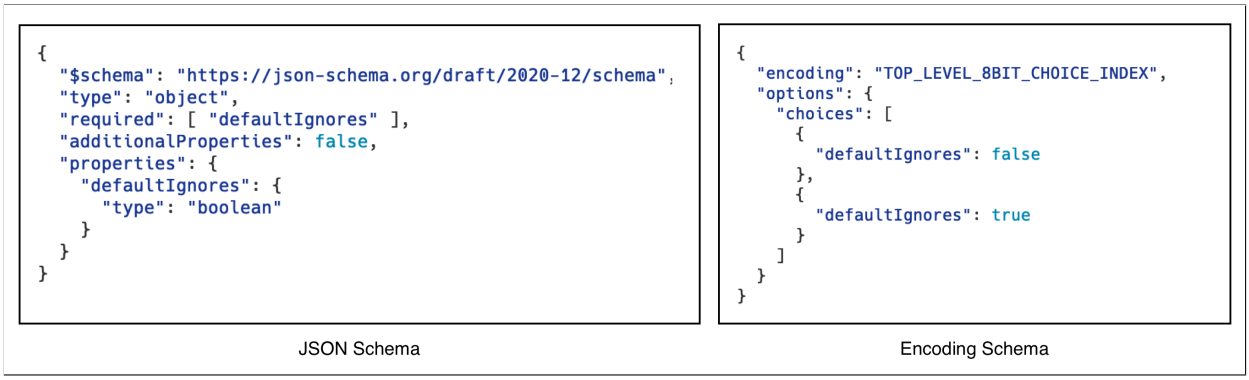

Figure 8.2: An example of an JSON BinPack Encoding Schema. The left side of the figure shows a JSON Schema [159] definition for a JSON [46] object with a single boolean property. The right side of the figure shows the corresponding JSON BinPack Encoding Schema, which flattens the finite possible states of the original schema definition into an 8-bit enumeration that consists of two values. The `encoding` property refers to an encoding defined by the JSON BinPack Encoder component discussed in subsection 8.3.3. The `TOP_LEVEL_8BIT_CHOICE_INDEX` encoding shown in this example and its options are summarized in Table A.2.

8.3.1.4 Core Components

Figure 8.1 shows a C4 [23] diagram that presents the architecture of JSON BinPack. The schema refinement phase of JSON BinPack is split into canonicalization and mapping parts. The following sections introduce the three core components of JSON BinPack: the *canonicalizer*, the *encoder* and the *mapper*.

8.3.2 Canonicalizer

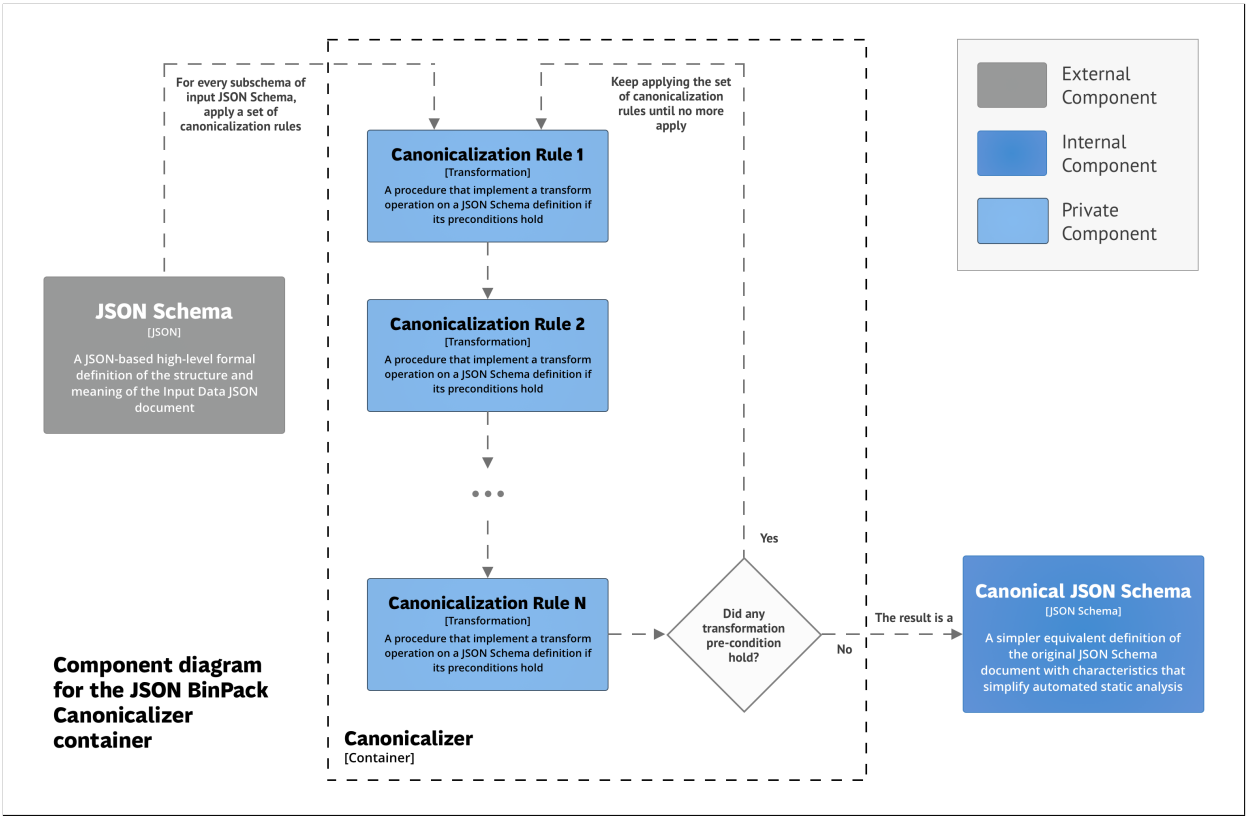

Figure 8.3: A component diagram based on the C4 software architecture visualization notation [23] for the JSON BinPack Canonicalizer container. The private components represent the set of formal transformation rules defined in the Canonicalizer. The C4 container diagram that introduces this component is presented in Figure 8.1.

In subsection 8.2.1, I propose JSON BinPack as a schema-driven serialization format and in subsection 8.2.3, I propose the adoption of JSON Schema [159] as its schema language. As discussed in subsubsection 3.2.3.1, JSON Schema is a particularly expressive and complex schema language. To mitigate such complexity in the context of static analysis, the *Canonicalizer* is a total function that maps JSON Schema definitions to equivalent but simpler JSON Schema definitions according to a set formal transformations. The concept of simplifying JSON Schema definitions based on formal transformations for static analysis purposes was originally introduced by [68]. I extend their work by modernizing and extending their set of canonicalization rules. Figure 8.3 shows a C4 [23] component diagram for the Canonicalizer container. The canonicalizer repeatedly applies the set of defined canonicalization transformations rules to every subschema of a given JSON Schema definition until no more transformations are possible. A JSON Schema definition that cannot be further modified by any canonicalization rule is considered to be a *Canonical JSON Schema*. In order to prevent an infinite loop in the canonicalization algorithm, canonicalization rules do not conflict with each other and the pre-condition of a given canonicalization rule does not hold after such canonicalization rule has been applied to the schema.

For example, using the notation proposed by [68], a canonicalization rule that removes duplicate branches from the `allOf` logical applicator specified in the Applicator vocabulary [159] discussed in section 3.2 is defined as follows:

$$\frac{\texttt{allOf} \in dom(s) \wedge \#s.\texttt{allOf} \neq \#\{x \mid s.\texttt{allOf}\}}{s \rightarrow s[\texttt{allOf} \mapsto \text{seq}\ \{x \mid s.\texttt{allOf}\}]} \tag{8.1}$$

The JSON Schema definition is denoted as s. The expression above the bar is the rule pre-condition and the expression below the bar is the rule transformation. The rule transformation is applied only if the pre-condition holds.

In this case, the `allOf` sequence is de-duplicated only if *s* declares the `allOf` keyword to a sequence that includes duplicate elements.

8.3.2.1 Objectives

The transformation rules defined in the Canonicalizer component are guided by the following objectives:

- **Avoid mixing keywords for multiple data types.** To simplify the production of schemas that describe union types, JSON Schema permits the schema-writer to declare keywords that operate on different data types in the same scope. At runtime, JSON Schema implementations must only take into consideration the keywords that apply to the instance data type. The canonicalization rules transform heterogeneous schemas into simpler homogeneous schemas combined using logic applicators.
- **Re-write advanced keywords in terms of simpler keywords.** Schema-writers may inadvertently produce schema definitions that make unnecessary use of advanced JSON Schema keywords. Whenever possible, I detect these cases and re-write the schemas using combinations of simpler keywords. For example, a complex JSON Schema definition that has a finite set of matching instances can be re-written as an enumeration.
- **Surface implicit constrains.** JSON Schema definitions may include implicit constraints that schema-writers typically do not explicitly define given the intrinsic properties of certain data types (such as integers) or the default values of certain JSON Schema keywords. These constrains are surfaced and explicitly defined in the schema.
- **Inline references to external schemas.** JSON Schema definitions may include external schema definitions that are typically resolved using the HTTP [55] protocol. Static analysis is simplified if the program can operate on the entire schema definition without requiring networking capabilities.

8.3.3 Encoder

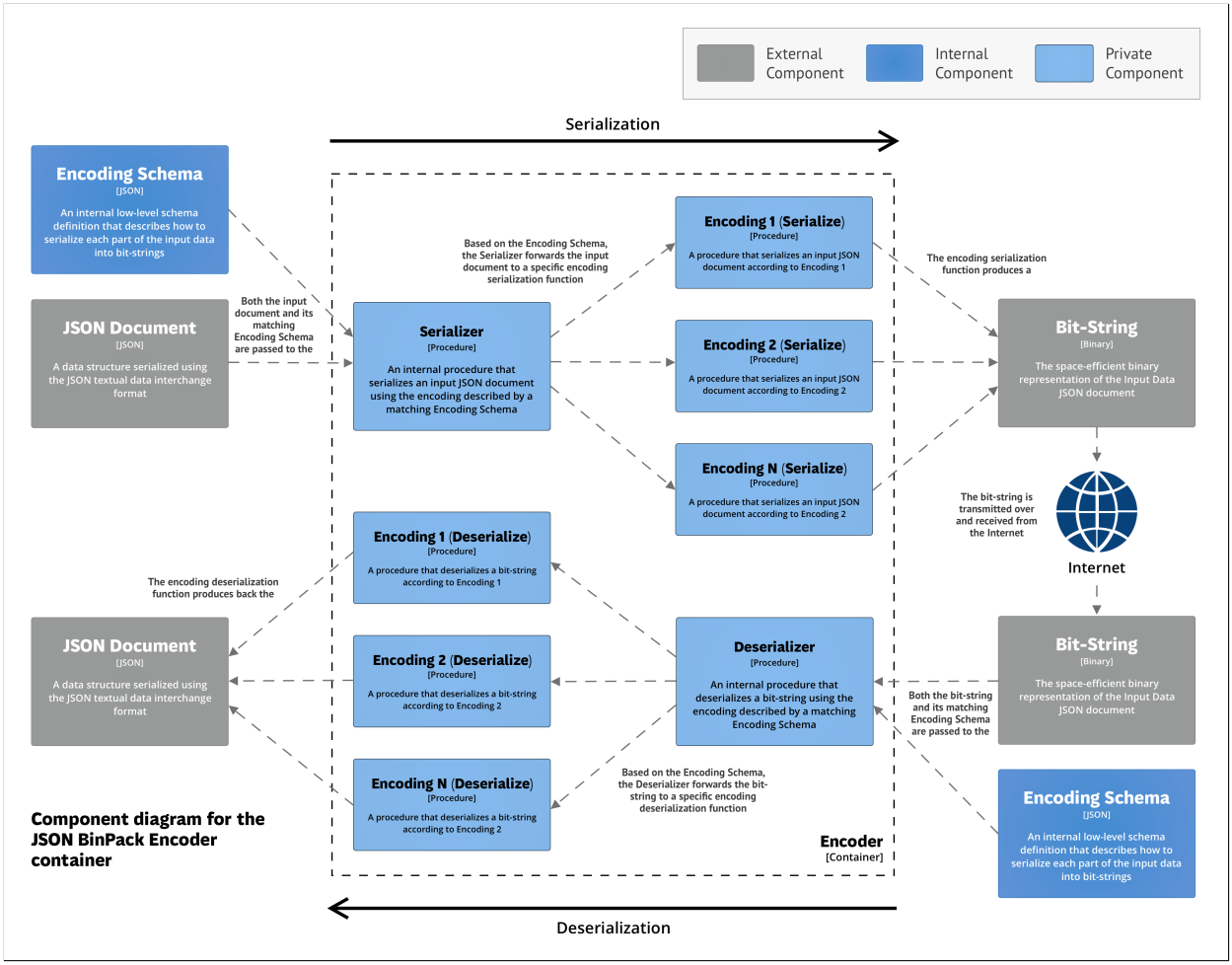

Figure 8.4: A component diagram based on the C4 software architecture visualization notation [23] for the JSON BinPack Encoder container. The C4 container diagram that introduces this component is presented in Figure 8.1.

In subsection 8.2.4, I make a case for the contextual nature of space-efficient data encodings and propose JSON BinPack to implement a diverse set of encodings that are space-efficient in different scenarios. The *Encoder* software component consists of a set of parameterized serialization and deserialization procedures targetting data types given different sets of contrains. In Table 8.2, I require that JSON BinPack shall be strictly JSON-compatible [46]. Therefore, the set of serialization and deserialization procedures defined in the *Encoder* software component are bounded by the JSON [46] data model. However, the set of serialization and deserialization procedures defined in the *Encoder* software component are agnostic to the schema language used to validate and describe the input data. Figure 8.4 shows a C4 [23] component diagram for the Encoder container. The Encoder defines a proxy serialization procedure that given a JSON document and a matching Encoding Schema, serializes the JSON document into a bit-string using the specific encoding implementation declared by the Encoding Schema. The deserialization process mirrors the serialization process through a proxy deserialization procedure that deserializes a bit-string into a JSON document using the specific encoding implementation declared by the Encoding Schema.

8.3.4 Mapper

The *Mapper* software component is the bridge between a Canonical JSON Schema produced by the Canonicalizer software component defined in subsection 8.3.2 and an Encoding Schema consumed by the Encoder software component defined in subsection 8.3.3. The Mapper statically analyses the input Canonical JSON Schema and de-

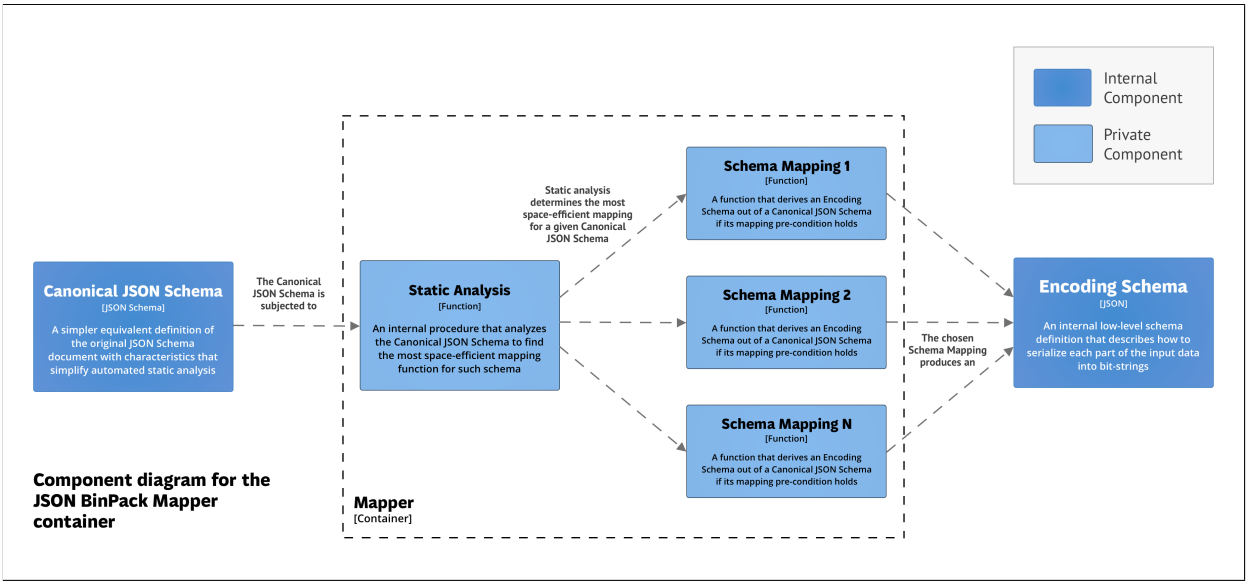

Figure 8.5: A component diagram based on the C4 software architecture visualization notation [23] for the JSON BinPack Mapper container. The C4 container diagram that introduces this component is presented in Figure 8.1.

terministically computes which encodings and respective encoding parameters defined by the Encoder are likely to produce the most space-efficient bit-strings for any JSON [46] instance that successfully validates against the given schema. Figure 8.5 shows a C4 [23] component diagram for the Mapper container.

For example, a mapping rule that associates a JSON Schema [159] definition that describes a bounded integer where the range of possible values is representable in 8-bits to a fictitious `UNSIGNED_INTEGER_8BIT` encoding that takes a lower bound parameter is defined as follows:

$$\frac{s.\texttt{type} = \mathit{integer} \wedge \{\texttt{minimum}, \texttt{maximum}\} \subseteq \mathit{dom}(s) \wedge s.\texttt{maximum} - s.\texttt{minimum} < 2^8}{\texttt{UNSIGNED\_INTEGER\_8BIT}(\langle \mathit{minimum} \rightsquigarrow s.\texttt{minimum} \rangle)} \tag{8.2}$$

The JSON Schema definition is denoted as s. The expression above the bar is the rule pre-condition and the expression below the bar is the encoding declaration.

8.4 Implementation

I developed an open-source proof-of-concept implementation of the JSON BinPack serialization specification hosted on GitHub [6] consisting of 37 canonicalization rules, 35 mapping rules and 34 parameterized encodings summarized in Appendix A that cover every data type supported by the JSON [46] data model. At this point in time, not many JSON Schema [159] implementations [7] support 2020-12, the latest draft discussed in section 3.2. AJV [8], the dominant and up-to-date JSON Schema implementation, is written to target the JavaScript [47] programming language. Therefore, I design the first JSON BinPack implementation around such ecosystem. The proof-of-concept implementation is developed using the Node.js v14.17.1 [9] JavaScript runtime, the TypeScript v4.2.4 [10] type-based JavaScript transpiler and the AJV v8.6.0 JSON Schema implementation. Figure 8.6 shows an example JavaScript [47] program that makes use of the proof-of-concept implementation. This proof-of-concept implementation is not production-ready. It is intended to give an early view of the potential of JSON BinPack to motivate the development of a more runtime-efficient and production-ready implementation.

```
const jsonbinpack = require('jsonbinpack');

// Declare a JSON Schema definition
const schema = {
  $schema: 'https://json-schema.org/draft/2020-12/schema',
  type: 'object',
  properties: {
    foo: { type: 'string' }
  }
};

// (1) Compile the JSON Schema definition into an Encoding Schema
const encodingSchema = await jsonbinpack.compileSchema(schema);

// (2) Serialize a matching JSON document using the Encoding Schema
const buffer = jsonbinpack.serialize(encodingSchema, {
  foo: 'bar'
});

// (3) Deserialize the bit-string into the original JSON document
const result = jsonbinpack.deserialize(encodingSchema, buffer);

// Print the resulting JSON document
console.log(JSON.stringify(result, null, 2));
```

Figure 8.6: An example JavaScript [47] program that uses the proof-of-concept JSON BinPack implementation to obtain an Encoding Schema and use it to perform serialization and deserialization. First (1), the example JSON Schema 2020-12 [159] definition is converted into an Encoding Schema as discussed in subsubsection 8.3.1.3 using the `jsonbinpack.compileSchema` function. Then (2), the resulting Encoding Schema is used to serialize an example JSON [46] document using the `jsonbinpack.serialize` function. Finally (3), the `jsonbinpack.deserialize` function is used to deserialize the resulting bit-string back into the original JSON document.

To showcase the inner workings of JSON BinPack, the test JSON [46] document introduced in subsection 5.1.3 is serialized using JSON BinPack and the resulting bit-strings are explained in Figure 8.7 and Figure 8.9 as done for other serialization specifications in the study presented in chapter 6. In Figure 8.7, the test JSON document is serialized using the JSON Schema [159] definition shown in Figure 8.8. Its corresponding Encoding Schema, which is too large to be included here, can be found on GitHub [11]. Such Encoding Schema makes use of 9 out of the 34

[6] https://github.com/jviotti/jsonbinpack
[7] https://json-schema.org/implementations.html
[8] https://ajv.js.org
[9] https://nodejs.org
[10] https://www.typescriptlang.org
[11] https://github.com/jviotti/jsonbinpack/blob/c029870287bedfa54afe7b4b5d39e2e95ecff91f/test/e2e/

encodings defined in Appendix A. In Figure 8.9, I showcase the schema-less mode of JSON BinPack produced by the encoding summarized in Table A.9 by serializing the test JSON document with the wildcard JSON Schema [159] definition `{}` that matches every instance.

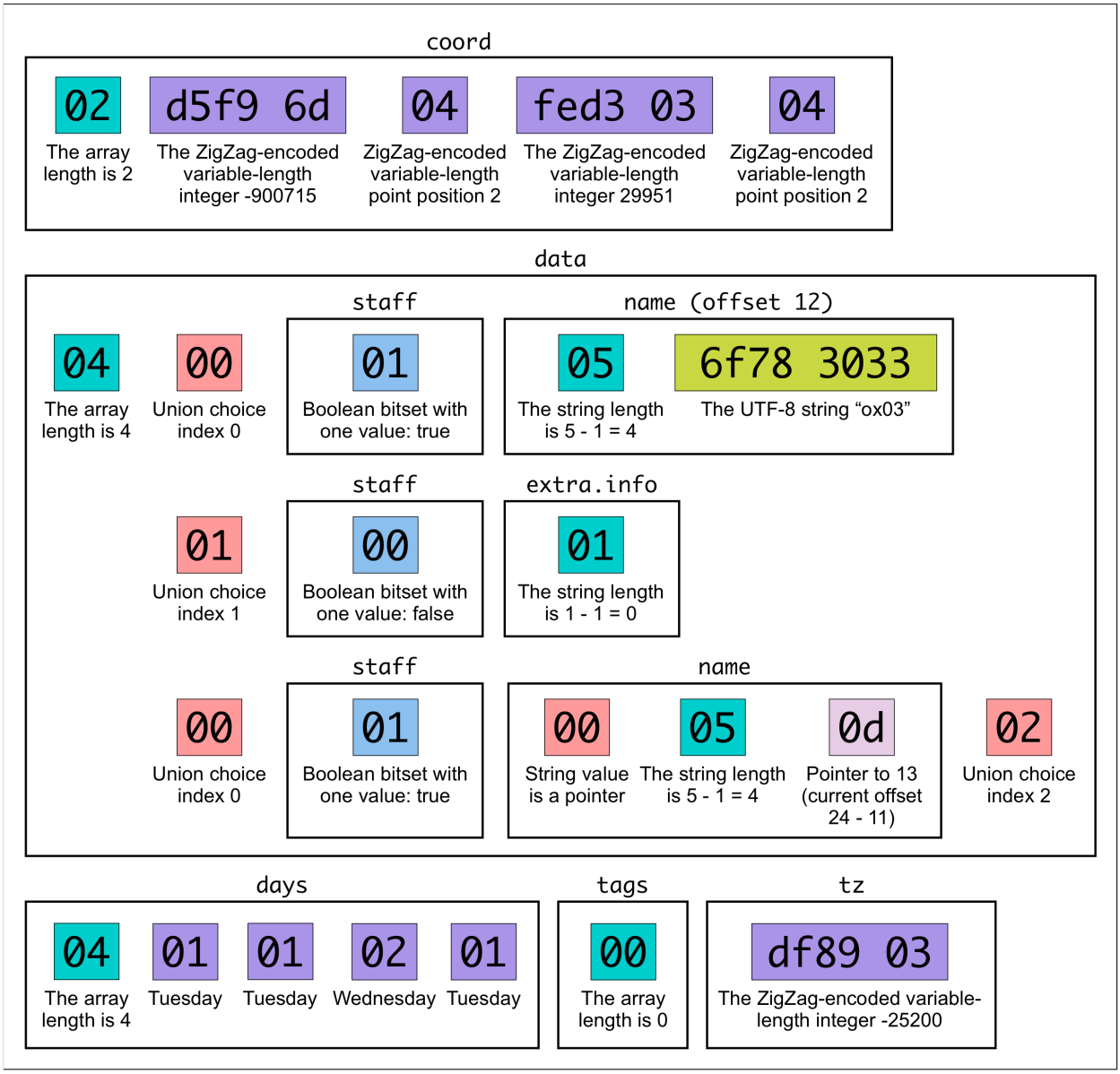

Figure 8.7: Annotated hexadecimal output of serializing the test input data introduced in subsection 5.1.3 using the JSON BinPack proof-of-concept implementation. The corresponding JSON Schema [159] definition used to produce this bit-string is shown in Figure 8.8. The encodings utilized are summarized in Appendix A.

`ox-test/schema-driven/encoding.json`

```json
{
  "type": "object",
  "required": [ "tags", "tz", "days", "coord", "data" ],
  "additionalProperties": false,
  "properties": {
    "tags": {
      "type": "array",
      "items": { "type": "string" }
    },
    "tz": { "type": "integer" },
    "days": {
      "type": "array",
      "items": {
        "type": "integer", "minimum": 0, "maximum": 6
      }
    },
    "coord": {
      "type": "array",
      "items": { "type": "number" }
    },
    "data": {
      "type": "array",
      "items": {
        "oneOf": [
          {
            "type": "object",
            "required": [ "name", "staff" ],
            "additionalProperties": false,
            "properties": {
              "name": { "type": "string" },
              "staff": { "type": "boolean" }
            }
          },
          {
            "type": "object",
            "required": [ "name", "staff", "extra" ],
            "additionalProperties": false,
            "properties": {
              "name": { "type": "null" },
              "staff": { "type": "boolean" },
              "extra": {
                "type": "object",
                "required": [ "info" ],
                "additionalProperties": false,
                "properties": {
                  "info": { "type": "string" }
                }
              }
            }
          },
          {
            "type": "object",
            "additionalProperties": false
          }
        ]
      }
    }
  }
}
```

Figure 8.8: The JSON Schema 2020-12 [159] definition used to serialize the test input data introduced in subsection 5.1.3 using the JSON BinPack proof-of-concept implementation. The resulting bit-string is shown in Figure 8.7.

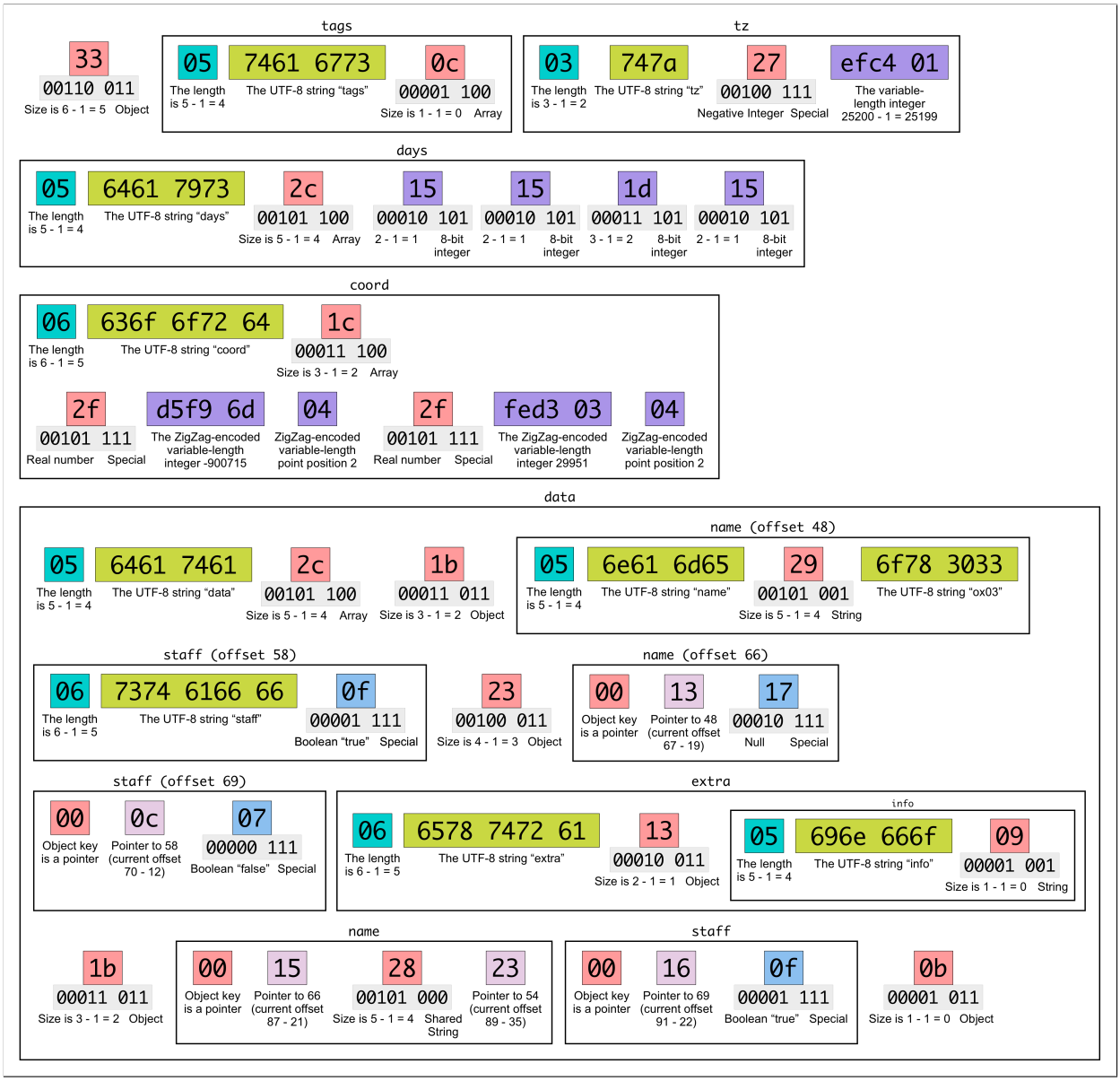

Figure 8.9: Annotated hexadecimal output of serializing the test input data introduced in subsection 5.1.3 using the JSON BinPack proof-of-concept implementation in schema-less mode by providing the `{}` wildcard JSON Schema [159] definition that matches every instance. The encodings utilized are summarized in Table A.10 and Table A.11.

8.5 Testing

The JSON BinPack serialization specification proof-of-concept implementation includes a comprehensive automated test suite [12] consisting of 1942 test cases categorized as unit tests, property tests and end-to-end tests. The entire test suite is executed for every commit using GitHub Actions [13] to support continuous integration.

8.5.1 Unit Testing

The JSON BinPack proof-of-concept implementation introduced in section 8.4 defines 616 unit tests that exercise the serialization specification at a functional level. These unit tests were implemented using the TypeScript [14] programming language and the *node-tap* open-source unit testing framework [15] while following the *Test Driven Development* [11] methodology. Refer to Figure 8.10 for an example unit test from the test suite [16] of integer encodings of the encoder component introduced in subsection 8.3.3.

```
tap.test('ARBITRARY_MULTIPLE_ZIGZAG_VARINT: should encode -25200 as 0xdf 0x89 0x03', (test) => {
  const context: EncodingContext = getDefaultEncodingContext()
  const buffer: ResizableBuffer = new ResizableBuffer(Buffer.allocUnsafe(3))
  const bytesWritten: number =
    ARBITRARY_MULTIPLE_ZIGZAG_VARINT(buffer, 0, -25200, { multiplier: 1 }, context)
  test.strictSame(buffer.getBuffer(), Buffer.from([ 0xdf, 0x89, 0x03 ]))
  test.is(bytesWritten, 3)
  test.end()
})
```

Figure 8.10: An example unit test that asserts that serializing the negative integer -25200 as a Little Endian Base 128 (LEB128) variable-length ZigZag-encoded [151] signed integer using the JSON BinPack `ARBITRARY_MULTIPLE_ZIGZAG_VARINT` encoding summarized in Table A.1 results in the 3-byte bit-string `0xdf 0x89 0x03`. The assertions executed by the unit test are highlighted.

8.5.2 Property-based Testing

[35] introduce QuickCheck, a tool to automate the process of generating test cases using the Haskell [94] functional programming language based on a formal definition of a functional property. QuickCheck generates a finite amount of input data based on a formal specification to automatically explore the problem space and assert that the functional property holds in every tested case. This testing technique is referred to as *property-based testing* [57] and is considered a variant of software fuzzing [134]. An open-source property-based testing framework for the JavaScript [47] programming language inspired by QuickCheck called *fast-check* [17] is used to define and test 27 formal properties covering the JSON BinPack serialization and deserialization functions of all supported data types. Refer to Figure 8.11 for an example property-based test from the test suite [18] of the encoder component introduced in subsection 8.3.3.

Using this technique highlighted several types of issues on the encoder component early on in the development phase:

- Offset arithmetic errors detected by randomizing the start offset of the buffer to serialize to and deserialize from.
- Buffer overflows detected by generating large input data.

[12] https://github.com/jviotti/jsonbinpack/tree/9d7b533012a38476ee359cdb76ec4b1e180051f7/test
[13] https://github.com/features/actions
[14] https://www.typescriptlang.org
[15] https://node-tap.org
[16] https://github.com/jviotti/jsonbinpack/blob/9d7b533012a38476ee359cdb76ec4b1e180051f7/test/encoder/integer/encode.spec.ts#L217-L225
[17] https://github.com/dubzzz/fast-check
[18] https://github.com/jviotti/jsonbinpack/blob/3cb625e8842462cb475ea4cdbf18c56b39a051bd/test/encoder/any.spec.ts#L135-L157

```
tap.test('ANY_PACKED_TYPE_TAG_BYTE_PREFIX: scalars', (test) => {
  fc.assert(fc.property(fc.nat(10), fc.oneof(
    fc.constant(null),
    fc.boolean(),
    fc.integer(),
    fc.float(),
    fc.double(),
    fc.string({
      maxLength: 1000
    })
  ), (offset: number, value: JSONValue): boolean => {
    const context: EncodingContext = getDefaultEncodingContext()
    const buffer: ResizableBuffer = new ResizableBuffer(Buffer.allocUnsafe(2048))
    const bytesWritten: number =
        ENCODE_ANY_PACKED_TYPE_TAG_BYTE_PREFIX(buffer, offset, value, {}, context)
    const result: AnyResult = DECODE_ANY_PACKED_TYPE_TAG_BYTE_PREFIX(buffer, offset, {})
    return bytesWritten > 0 &&
      result.bytes === bytesWritten && result.value === value
  }), {
    verbose: false
  })

  test.end()
})
```

Figure 8.11: A simple property-based test that asserts that for an infinite set of scalar JavaScript [47] values, serializing and deserializing the input data using the `ANY_PACKED_TYPE_TAG_BYTE_PREFIX` encoding summarized in Table A.9 at a random offset between 0 and 10 of a test buffer reads and writes the same amount of bytes and results in the original input data. The domain of the property and its assertion are highlighted.

- Numeric truncation and overflows on integer and real number encodings detected by generating large numeric values.
- Functional mismatches between serialization and deserialization function pairs for certain encodings detected with specific encoding parameters and combinations of input data.

8.5.3 End-to-End Testing

8.5.3.1 Input Data From Previous Research

I define automated end-to-end test cases that make use of the JSON BinPack public programming interface to generate an encoding schema and serialize and deserialize the 27 JSON [46] documents introduced in subsection 5.2.3 and the example analysis JSON [46] document introduced in subsection 5.1.3. Each of these documents are serialized in schema-driven and schema-less modes resulting in 56 end-to-end test cases. The test runner asserts that serializing and deserializing every input document results in the respective original input document.

8.5.3.2 The JSON Schema Official Test Suite

The JSON Schema organization maintains a publicly-available open-source test suite [19] to help JSON Schema implementations adhere to the Core [159] and Validation [160] JSON Schema specifications. The test suite consist of a set of contrived schema that showcase features of the official JSON Schema vocabularies for every release of JSON Schema. Refer to Figure 8.12 for an example test case [20]. The test suite that targets the latest version of JSON Schema at commit `eaa5bff` [21] is organized as a set of 47 required and 24 optional suites defining a total of 326 schemas associated with a total of 1331 instances that must either validate or fail against the respective schemas.

[19] https://github.com/json-schema-org/JSON-Schema-Test-Suite

[20] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/allOf.json#L218-L243

[21] https://github.com/json-schema-org/JSON-Schema-Test-Suite/tree/eaa5bffc22658ebc96bb0f3f044fca8be82afc63

```
{
    "description": "nested allOf, to check validation semantics",
    "schema": {
        "allOf": [
            {
                "allOf": [
                    {
                        "type": "null"
                    }
                ]
            }
        ]
    },
    "tests": [
        {
            "description": "null is valid",
            "data": null,
            "valid": true
        },
        {
            "description": "anything non-null is invalid",
            "data": 123,
            "valid": false
        }
    ]
}
```

Figure 8.12: An example test case that covers how the `allOf` applicator keyword from the Core [159] must behave when nested. The JSON Schema definition provided in the test case and the associated instances are highlighted.

The JSON BinPack proof-of-concept implementation uses the official JSON Schema test suite to derive 1172 automated end-to-end test cases. The JSON BinPack test suite generates encoding schemas from every schema definition provided by the JSON Schema official test suite and uses the resulting encoding schemas to serialize and deserialize any declared instance that validates against the given schemas. The JSON BinPack proof-of-concept implementation uses 63 out of the 71 suites provided by the official JSON Schema tests. The `dynamicRef` [22] and `float-overflow` [23] suites are omitted as they crash the latest version of one of the external dependencies of the Canonicalizer component at the time of this writing [24]. JSON Schema defines the `format` keyword from the Validation specification [160] in two mutually incompatible vocabularies that affect the semantics of such keyword and expects implementations to choose one: `format-annotation` and `format-assertion`. The former vocabulary defines the `format` keyword as an annotation that takes no effect during validation. Therefore, I choose to support the latter and not test the `format-annotation` suite [25]. For simplicity in the implementation of the JSON BinPack proof-of-concept test runner, the JSON Schema test suites that require resolving external schemas over the HTTP [55] protocol are not taken into account. This includes the `refRemote` [26], `ref` [27], `id` [28], `anchor`

[22] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/dynamicRef.json

[23] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/optional/float-overflow.json

[24] https://apitools.dev/json-schema-ref-parser/

[25] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/format.json

[26] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/refRemote.json

[27] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/ref.json

[28] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/id.json

[29] and `unknownKeyword` [30] suites.

[29] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/anchor.json

[30] https://github.com/json-schema-org/JSON-Schema-Test-Suite/blob/eaa5bffc22658ebc96bb0f3f044fca8be82afc63/tests/draft2020-12/unknownKeyword.json

9 | A Benchmark of JSON BinPack

9.1 Methodology

To evaluate the JSON BinPack schema-driven binary serialization specification against the requirements described in section 8.1, I benchmark the JSON BinPack proof-of-concept implementation introduced in section 8.4 using the open-source space-efficiency benchmark framework for JSON-compatible binary serialization specifications implemented as part of chapter 7. The approach is the following:

1. **Include JSON BinPack.** Extend the open-source automated benchmark software implemented as part of chapter 7 to recognize JSON BinPack as a JSON-compatible binary serialization specification.

2. **Schema Definitions.** Write JSON Schema [159] definitions for the 27 input JSON [46] documents listed in subsection 5.2.3. As discussed in subsubsection 3.2.3.2, JSON Schema [159] definitions are as strict or loose as the schema-author decides. For this reason, two JSON Schema definitions are written for each of the 27 JSON [46] documents: a *strict* and a *loose* schema.

3. **Benchmark.** Use the automated benchmark software to serialize and de-serialize the 27 real-world JSON [46] documents listed in subsection 5.2.3 with the JSON BinPack binary specification using the strict and loose JSON Schema [159] definitions discussed in the previous step. section 1.5 proposes the idea of schema-less as subset of schema-driven. Under this concept, I consider serializing the input data using a loose schema definition that matches any instance as executing the JSON BinPack binary specification in *schema-less* mode. Conversely, I consider serializing the input data using the strict schema definition as executing the JSON BinPack binary specification in *schema-driven* mode.

4. **Results.** Provide plots to visualize the benchmark results of JSON BinPack in comparison to the 14 JSON-compatible serialization specifications and encodings listed in section 5.2.

5. **Conclusions.** Discuss the benchmark results and how JSON BinPack compares to the 14 JSON-compatible serialization specifications and encodings listed in section 5.2 in terms of space-efficiency and JSON-compatibility.

9.2 Benchmark

This section presents bechmark results for the 27 JSON [46] documents considered by chapter 7. For each plot, the schema-driven serialization specifications that produce the smallest bit-strings for the corresponding document are highlighted in red and the schema-less serialization specifications that produce the smallest bit-strings for the corresponding document are highlighted in orange. *JSON BinPack (Schema-driven)* corresponds to executing JSON BinPack with a strict schema definition and *JSON BinPack (Schema-less)* corresponds to executing JSON BinPack with the wildcard schema definition that matches any instance. In the interest of brevity, I provide condensed bar plots that do not take compression into account. These plots are shown in Figure 9.1, Figure 9.2, Figure 9.3 and Figure 9.4 in groups of 8. For the interested reader, the detailed plots that take compression into account can be found in the open-source benchmark results published on GitHub [1].

[1] https://github.com/jviotti/binary-json-size-benchmark

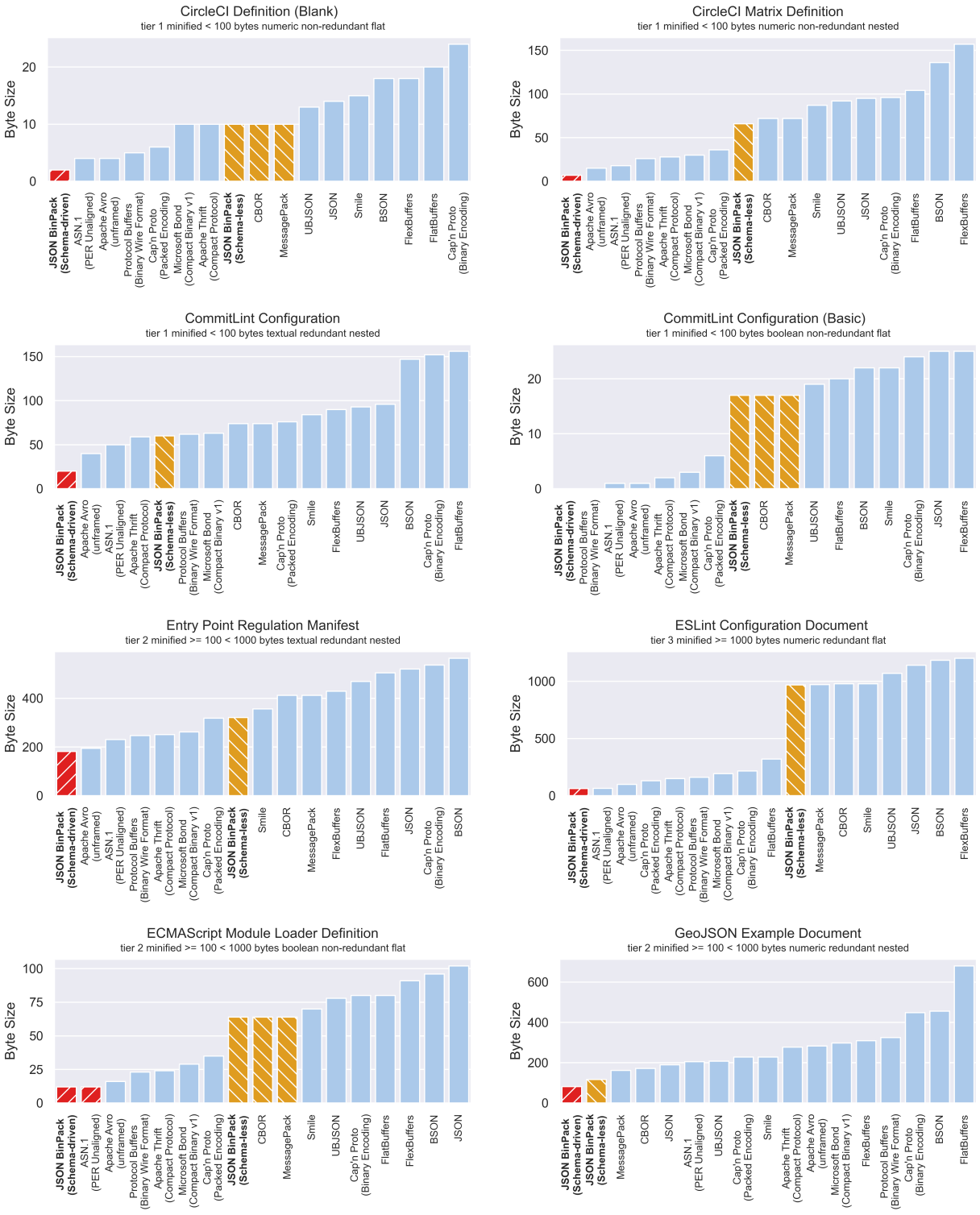

Figure 9.1: The benchmark results for the first set of input data listed in subsection 5.2.3.

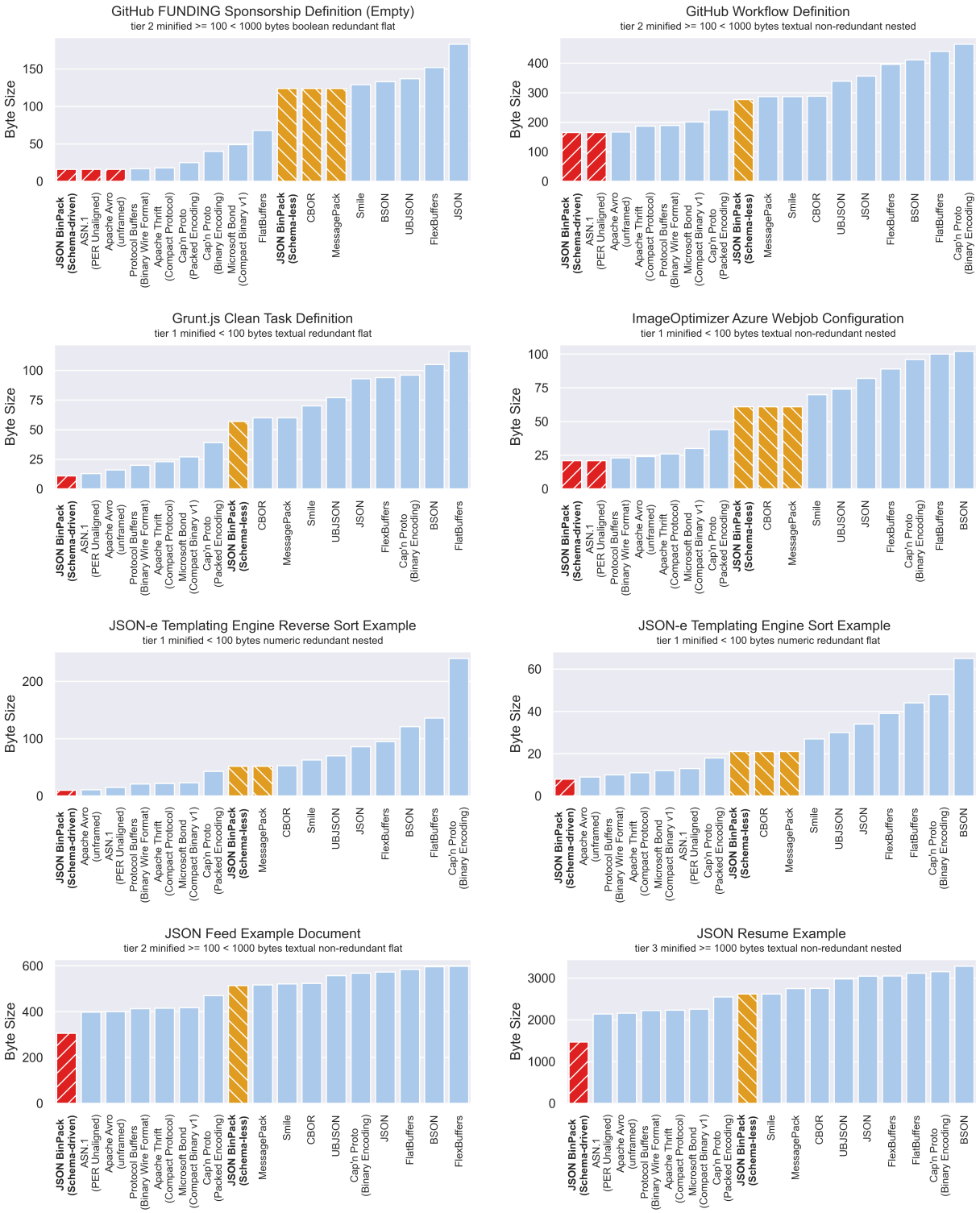

Figure 9.2: The benchmark results for the next set of input data listed in subsection 5.2.3.

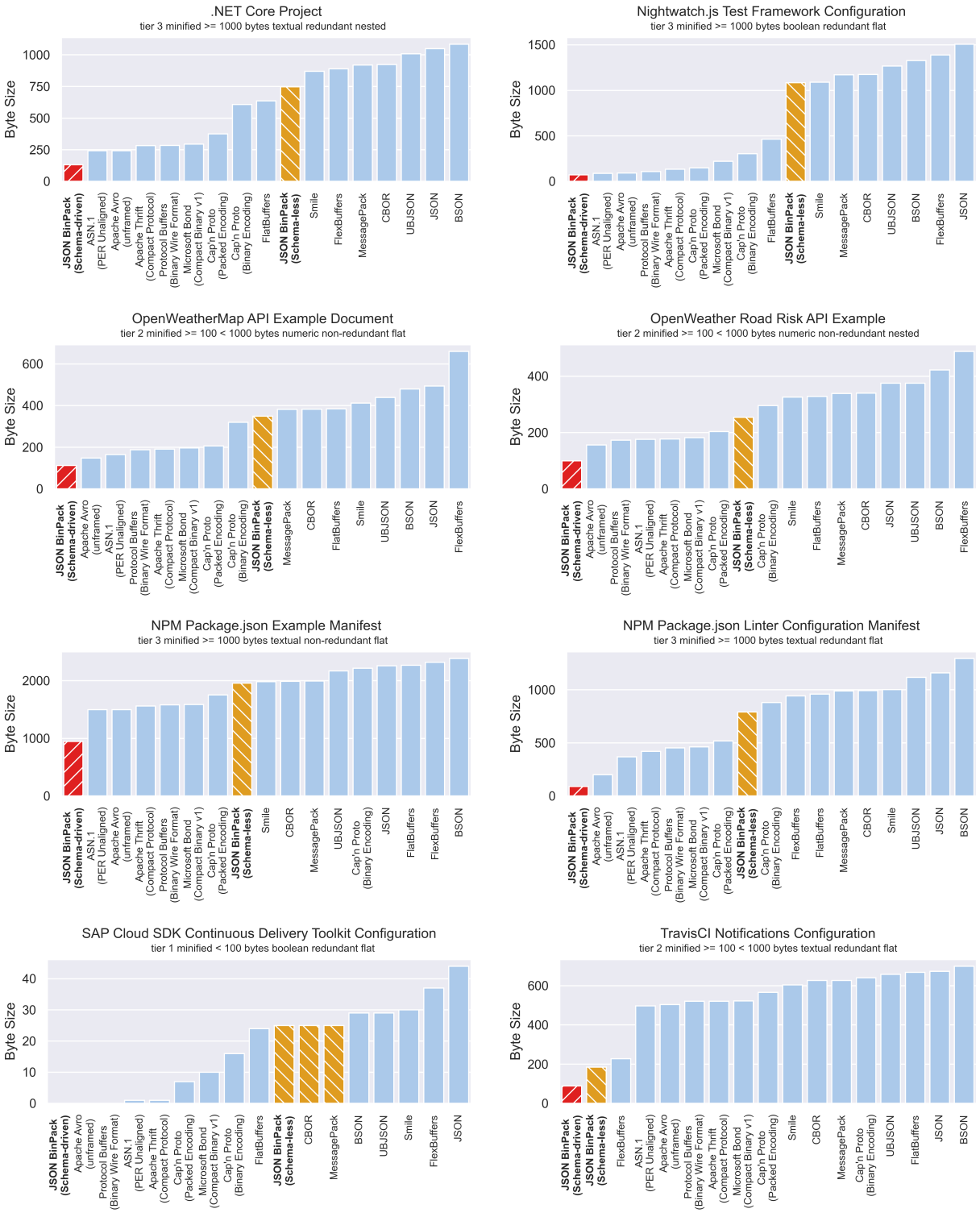

Figure 9.3: The benchmark results for the next set of input data listed in subsection 5.2.3.

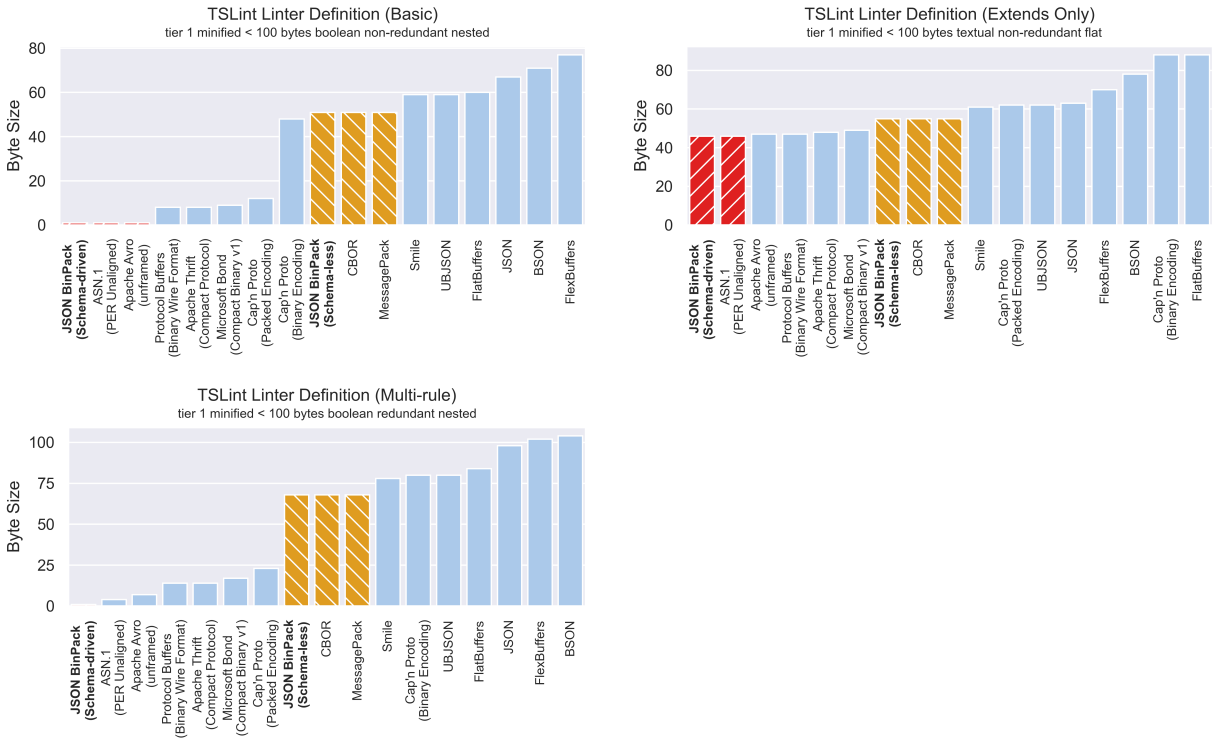

Figure 9.4: The benchmark results for the final set of input data listed in subsection 5.2.3.

9.3 Discussion

Table 9.1: A summary of the size reduction provided by the JSON BinPack binary serialization specification proof-of-concept implementation introduced in section 8.4 in both schema-driven and schema-less mode in comparison to JSON [46] given the input data listed in subsection 5.2.3.

Serialization Specification	**Size Reductions in Comparison To JSON**					**Negative Cases**
	Maximum	**Minimum**	**Range**	**Median**	**Average**	
JSON BinPack (Schema-driven)	100%	26.9%	73	86.7%	78.7%	0 / 27 (0%)
JSON BinPack (Schema-less)	72.5%	10.2%	62.2	30.6%	30.5%	0 / 27 (0%)
Averages	**86.2%**	**18.6%**	**67.6**	**58.6%**	**54.6%**	**0%**

9.3.1 JSON BinPack (Schema-driven) in Comparison to Uncompressed JSON

As demonstrated in section 9.2, the JSON BinPack schema-driven binary serialization specification, denoted as *JSON BinPack (Schema-driven)*, is as space-efficient or more space-efficient than every other serialization specification for the 27 proposed input data considered by chapter 7. Unlike any other considered binary serialization specification, JSON BinPack is strictly space-efficient in comparison to JSON [46] given the input data.

In chapter 7, I find that the most space-efficient JSON-compatible binary serialization specifications are ASN.1 PER Unaligned [124] and Apache Avro [61]. ASN.1 PER Unaligned [124] results in 71.4% and 65.7% median and average size reductions with a maximum of 98.5% and a minimum of negative 7.9%. Apache Avro [61] results in 73.5% and 65.7% median and average size reductions with a maximum of 100% and a minimum of negative 48.9%. In comparison, JSON BinPack produces strictly space-efficient results with a smaller range and no negative

cases: 86.7% and 78.7% median and average size reductions with a maximum of 100% and a minimum of 26.9% as shown in Table 9.1. Additionally, JSON BinPack provides improvements in terms of space-efficiency in comparison to the best-performing schema-driven binary serialization specifications for documents that are highly redundant or nested according to the JSON [46] taxonomy introduced in chapter 4. For example, JSON BinPack produces bit-strings that are 82%, 75% and 60% smaller than the best-performing schema-driven alternatives for *TravisCI Notifications Configuration* (see Figure 9.3), *TSLint Linter Definition (Multi-rule)* (see Figure 9.4) and *GeoJSON Example Document* (see Figure 9.1), respectively.

JSON BinPack matches but not increases the size reduction characteristics provided by the most space-efficient serialization specifications in 8 out of the 27 cases. In terms of the taxonomy for JSON [46] documents introduced in chapter 4, this list includes 5 out of the 7 documents considered boolean and 3 out of the 6 documents considered textual and non-redundant. These 8 documents represent cases where ASN.1 PER Unaligned [124], Apache Avro [61] and Protocol Buffers [67] perform at or close to the optimal level in terms of space-efficiency. For example, JSON BinPack and Protocol Buffers [67], and JSON BinPack, Apache Avro [61] and Protocol Buffers [67], serialize the *CommitLint Configuration (Basic)* Tier 1 Minified $<$ 100 bytes, boolean, non-redundant and flat (see Figure 9.1) and the *SAP Cloud SDK Continuous Delivery Toolkit Configuration* Tier 1 Minified $<$ 100 bytes, boolean, redundant and flat (see Figure 9.3) JSON [46] documents, respectively, into 0-byte bit-strings.

9.3.2 JSON BinPack (Schema-less) in Comparison to Uncompressed JSON

As explained in section 9.1, I also benchmark JSON BinPack in schema-less mode by serializing every input JSON [46] document with a *loose* JSON Schema [159] definition that matches every instance. As shown in section 9.2, the schema-less mode of the JSON BinPack binary serialization specification, denoted as *JSON BinPack (Schema-less)*, is as space-efficient or more space-efficient than every other schema-less serialization specification considered by chapter 7 for the 27 proposed input data. Like CBOR [17] and MessagePack [64], JSON BinPack in schema-less mode is strictly space-efficient in comparison to JSON [46]. However, JSON BinPack in schema-driven mode is strictly space-efficient in comparison to JSON BinPack in schema-less mode.

In chapter 7, I find that the most space-efficient JSON-compatible binary schema-less serialization specifications are CBOR [17] and MessagePack [64]. CBOR [17] results in 22.5% and 22.4% median and average size reductions with a maximum of 43.2% and a minimum of 6.8%. Similarly, MessagePack [64] results in 22.7% and 22.8% median and average size reductions with a maximum of 43.2% and a minimum of 6.8%. In comparison, JSON BinPack in schema-less mode produces strictly space-efficient results: 30.6% and 30.5% median and average size reductions with a maximum of 72.5% and a minimum of 10.2% as shown in Table 9.1. Additionally, JSON BinPack in schema-less mode provides significant improvements in terms of space-efficiency in comparison to the best-performing schema-less binary serialization specifications for documents that have a high-degree of nesting according to the JSON [46] taxonomy defined in chapter 4. For example, JSON BinPack produces bit-strings that are 27.7%, 22% and 18.9% smaller than the best-performing schema-less alternatives for *GeoJSON Example Document* (see Figure 9.1), *OpenWeather Road Risk API Example* (see Figure 9.3) and *CommitLint Configuration* (see Figure 9.1), respectively. Furthermore, JSON BinPack in schema-less mode produces space-efficient results in comparison to every schema-driven binary serialization specification for *GeoJSON Example Document* (see Figure 9.1) and *TravisCI Notifications Configuration* (see Figure 9.3), only second to JSON BinPack executed in schema-driven mode.

JSON BinPack in schema-less mode matches but not increases the size reduction characteristics provided by the most space-efficient schema-less serialization specifications in 11 out of 27 cases. In terms of the taxonomy for JSON [46] documents defined in chapter 4, this list includes 9 out of the 12 documents considered Tier 1 Minified $<$ 100 bytes and 2 out of the 2 documents considered Tier 2 Minified $\geq$ 100 $<$ 1000 bytes and boolean. These 11 documents represent cases where both CBOR [17] and MessagePack [64] perform close to the optimal level in terms of space-efficiency for a schema-less serialization specification.

9.3.3 JSON BinPack in Comparison to Compressed JSON

In chapter 7, I conclude that general-purpose data compression tends to yield negative results for JSON [46] documents that are Tier 1 Minified $<$ 100 bytes according to the proposed taxonomy given that the auxiliary data structures encoded by dictionary-based compressors may exceed the size of such small input documents. However, leaving Tier 1 Minified $<$ 100 bytes documents aside, best-case compressed JSON [46] is space-efficient in comparison to the considered schema-less binary serialization specifications in 86.6% of the cases. Leaving Tier 1 Minified $<$ 100

Table 9.2: A summary of the size reduction provided by the JSON BinPack binary serialization specification proof-of-concept implementation introduced in section 8.4 in both schema-driven and schema-less mode in comparison to the best case scenarios of compressed JSON [46] given the compression formats listed in subsection 5.2.6 and the input data listed in subsection 5.2.3.

Serialization Specification	**Size Reductions in Comparison To Compressed JSON**					**Negative Cases**
	Maximum	**Minimum**	**Range**	**Median**	**Average**	
JSON BinPack (Schema-driven)	100%	5.65%	94.3	76.1%	66.8%	0 / 27 (0%)
JSON BinPack (Schema-less)	69.6%	-146.4%	216.1	7.4%	-5.27%	13 / 27 (48.1%)
Averages	**84.4%**	**-70.3%**	**155.2**	**41.7%**	**30.7%**	**24%**

bytes documents aside, best-case compressed JSON [46] is strictly space-efficiency in comparison to the considered schema-driven binary serialization specifications in 33.3% of the cases.

While JSON BinPack in schema-less mode matches or outperforms the alternative schema-less binary serialization specifications considered in chapter 7 as shown in subsection 9.3.2, best-case compressed JSON [46] is space-efficient in comparison to JSON BinPack in schema-less mode in 13 out of the 27 considered cases as shown in Table 9.2. Of these 13 negative cases, 8 documents are considered textual according to the taxonomy defined in chapter 4. However, unlike the other considered schema-driven binary serialization specifications, JSON BinPack in schema-driven mode is space-efficient in comparison to best-case compressed JSON [46] as shown in Table 9.2 in terms of the median and average with size reductions of 76.1% and 66.8%, respectively. Existing literature [66] [4] show that compressed textual schema-less serialization specifications such as JSON [46] can outperform compressed and uncompressed schema-driven binary serialization specifications in terms of space-efficiency. However, I conclude that a space-efficient schema-driven serialization specification such as JSON BinPack can outperform general-purpose data compression.

10 | Schema Evolution

10.1 The Problem of Schema Compatibility

Schema evolution is the problem of updating a schema definition while ensuring that the programs relying on it can continue to operate. The study of schema evolution originated in the context of relational databases to evolve database schemas without disrupting client applications [118]. In the context of serialization specifications, schema evolution is concerned with how bit-string producers and bit-string consumers can intercommunicate despite future updates to the structure of the bit-strings they are concerned with.

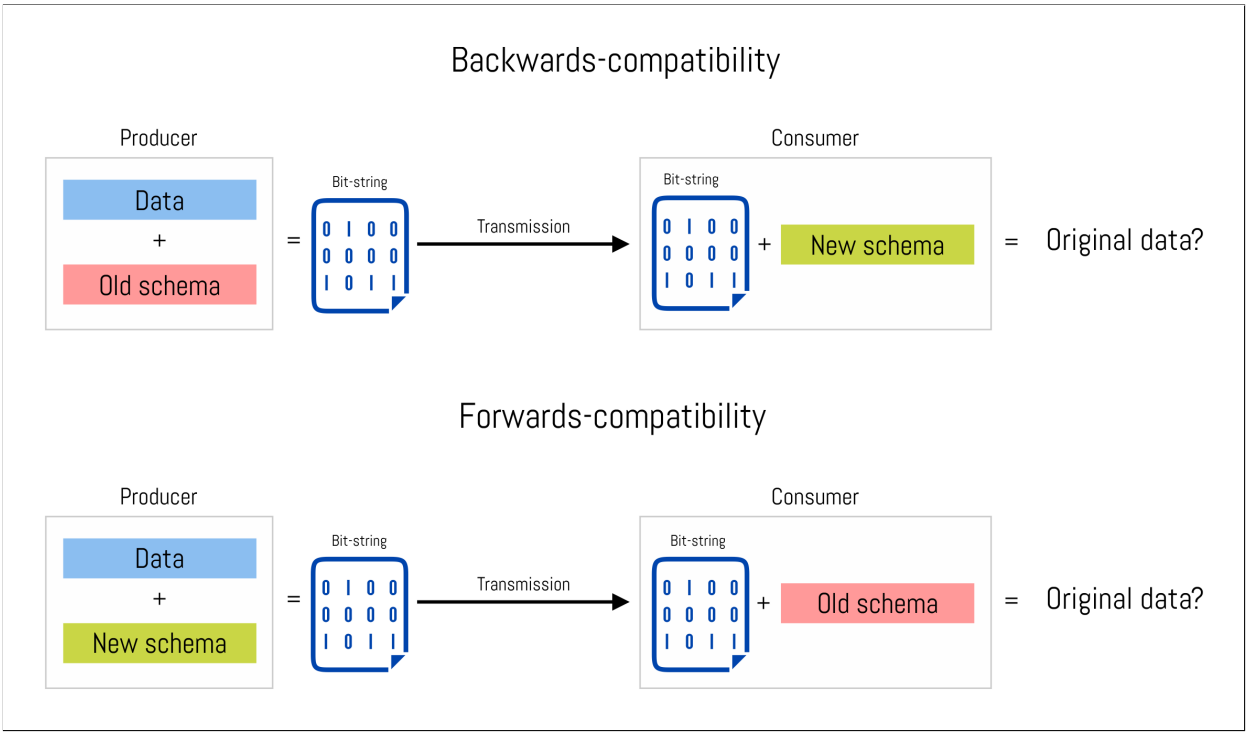

Figure 10.1: In both of these cases, the producer serializes a data structure using one version of the schema and the consumer attempts to deserialize the resulting bit-string using another version of the schema. Schema evolution is concerned with whether the consumer will succeed in obtaining the original data structure or not.

As discussed in section 1.4, programs using schema-driven serialization specifications must know in advance the schema definitions of the messages they are expecting to exchange. This problem is exacerbated by the fact that schema definitions are typically updated in response to new or changing requirements. [98] state that software requirements continuously change and as a result of these changes software projects tend to fail. Schema evolution is an important topic in the context of schema-driven serialization specifications as updating a schema definition may result in a risky and expensive operation that requires re-compilation and coordinated re-deployment of all the programs relying on such schema.

Two schemas are *compatible* if one schema can deserialize a bit-string produced by the other schema and recover the original information. There are three levels of schema compatibility:

- **Backwards.** The first schema is backwards-compatible with respect to the second schema if the first schema can deserialize data produced by the second schema.
- **Forwards.** The first schema is forwards-compatible with respect to the second schema if the second schema can deserialize data produced by the first schema.
- **Full.** The new schema is fully-compatible with respect to the old schema if it is both backwards and forwards compatible with respect to the old schema.

We can think of a schema as a set of its valid instances where the following rules apply:

- A backwards or forwards compatible transformation to the schema *expands* or *confines* the set of its valid instances, respectively. A fully-compatible transformation to the schema keeps the set of its valid instances intact.
- A schema transformation results in an incompatible schema if neither of the sets is a subset of the other.
- A schema is *fully-compatible* with respect to itself.
- The first schema is *backwards-compatible* with respect to the second schema if and only if the second schema is *forwards-compatible* with respect to the first schema.
- Compatibility is a transitive property. A schema is transitive backwards, transitive forwards, or transitive fully-compatible with respect to a set of schemas if it is backwards, forwards, or fully compatible with respect to each of the schemas in the set, respectively.

10.2 Deploying Schema Transformations

Consider a system involving a set of consumers and a set of producers that exchange information using a schema-driven serialization specification. In such a system, the rules for deploying *compatible* schema transformations with zero-downtime are as described in Table 10.1. Deploying *incompatible* schema transformations typically involves re-deploying all consumers and producers at the same time or including multiple incompatible schemas in each of the programs and adding application-specific logic to decide which schema to use when.

The same program in the system might be both a producer and a consumer. In this case, consider the program to use one schema to produce data and another schema to consume data where the two schemas may be equal. Therefore, each of the schemas within the same program can be deployed separately using the rules described in Table 10.1.

Table 10.1: These are the rules for deploying compatible schema transformations with zero-downtime. For example, it is safe to deploy a forwards-compatible schema transformation to any producer. However, deploying a forwards-compatible schema transformation to any consumer requires first deploying the schema transformation to all producers.

	Backwards-compatible schema transformation	**Forwards-compatible schema transformation**	**Fully-compatible schema transformation**
Deploy to Producer	Deploy transformation to consumers first	Safe	Safe
Deploy to Consumer	Safe	Deploy transformation to producers first	Safe

10.3 Types of Compatibility Resolution

Every schema-driven serialization specifications discussed in this study allows the schema-writer to perform certain schema transformations in compatible manners. I found that we can categorize schema-driven serialization specifications into two groups based on how they approach schema compatibility resolution: *data-based resolution* or *schema-based resolution*.

Data-based resolution. Every schema-driven serialization specification discussed in this study except for Apache Avro [61] falls into this category. In this type of schema compatibility resolution, the serialization specification tries to understand the data by deserializing the bit-string as if it was produced with the new schema, accomodating to potential mismatches at runtime.

Schema-based resolution. This approach is pioneered by Apache Avro [61], which refers to it as *symbolic resolution*. In comparison to the other schema-driven serialization specifications analyzed in this study, an application deserializing an Apache Avro bit-string has to provide both the *exact schema* that was used to produce the bit-string and the new schema. The implementation attempts to resolve the differences between the schemas before deserializing the bit-string in order to determine how to adapt any instance to the new representation. The bit-string is deserialized

using the old schema and transformed to match the new schema. [152] briefly discusses the problem of integrating heterogenous JSON datasets by resolving differences at the schema-level using a similar approach. [156] propose a similar approach based on version control systems where the codebase only maintains the latest schema definition and code to upgrade older bit-strings to the latest version is auto-generated based on the project commit history.

Discussion. I found that implementations of *data-based resolution* schema-driven serialization specifications, with some exceptions, tend to perform little runtime checks to ensure data consistency, presumably for performance reasons. For example, if the schema declares that the piece of data to follow is a Little Endian 64-bit unsigned integer, then the deserialization specification may blindly try to interpret the next 64-bits of the bit-string as such, resulting in many cases in silently-incompatible unpredictable results rather than informative runtime exceptions. In comparison to *data-based resolution* schema-driven serialization specifications, I found that *schema-based resolution* tends to produce informative runtime exceptions rather than unpredictable silently-incompatible results. However, *schema-based resolution* specifications require the consumer to know the exact schema that was used to produce the data and have it available at the deserialization process which may result in more complicated schema transformation deployments.

Based on the schema evolution experiments performed in Appendix C, I conclude that none of these approaches produce specifications that are clearly more advantageous with regards to compatible schema transformations: with some specification-specific exceptions, most specifications tend to support the same compatible schema transformations.

10.4 JSON Schema Evolution

The problem of schema evolution in the context of JSON Schema [159] is not unique to data serialization. Any software system that uses JSON Schema [159] to define an interface based on a data structure for inter-operatiblity purposes faces the problem of schema evolution when making changes to such data structure. The 7 schema-driven binary serialization specification covered in [151] implement non-interchangeable custom schema languages that are only maintained for the purpose of data serialization within a given serialization specification. In comparison to these custom schema languages, JSON Schema [159] is a general purpose extensible schema language for JSON [46] documents with broad applicability across use cases and industries. As a consequence of such generality, JSON Schema features a vibrant community that implements open-source tooling [1] to tackle foundational problems such as schema evolution in a generic manner.

[68] and [63] study the problem of *schema containment* in the context of JSON Schema [159] evolution. This problem is concerned with checking whether a JSON Schema definition is a subset of another JSON Schema definition. While schema containment answers whether compatibility is preverved across multiple versions of a JSON Schema definition, schema containment does not solve the schema mapping problem: how an instance that matches a specific JSON Schema definition version can be automatically transformed to match a different schema version without losing its semantics. The Ink & Switch [2] research organization focused on distributed local-first software implements *Project Cambria* [3], an open-source software library that implements a schema-based resolution approach to schema evolution with support for JSON Schema [159]. In comparison to schema containment, Cambria uses the concept of a lens, a data structure that defines a bi-directional transformation between two other data structures as defined in [72], to associate two schema definitions. With Cambria, lenses are JSON [46] documents that are used to automatically transform JSON [46] documents that match a JSON Schema [159] definition into JSON [46] documents that match another JSON Schema [159] definition, and vice-versa. Refer to Figure 10.2 for an example. While Cambria implements a language to define lenses and an engine to apply a lens to a JSON Schema [159] definition, the schema-writer is expected to manually write the lenses that accurately associate the JSON Schema [159] definitions used by the given software system. To simplify the schema evolution process, I envision that Cambria can be extended to attempt to automatically derive a lens out of two versions of a JSON Schema [159] definition.

To avoid reinventing the wheel, I tap into the JSON Schema [159] ecosystem and propose that users of JSON BinPack adopt a readily available solution to schema evolution that implements schema-based resolution such as Project Cambria.

[1] https://json-schema.org/implementations.html
[2] https://www.inkandswitch.com
[3] https://www.inkandswitch.com/cambria/

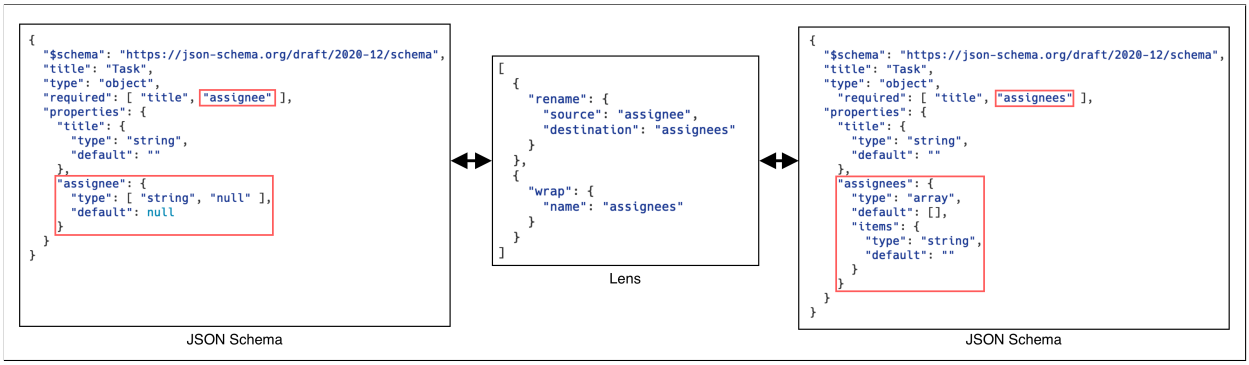

Figure 10.2: An example of using Cambria to define a lens that associates two JSON Schema definitions adapted from the Cambria research essay. The lens data structure is used to convert any JSON [46] document that matches the JSON Schema [159] definition at the left to a JSON [46] document that matches the JSON Schema [159] definition at the right, and vice-versa.

11 | Reflections

11.1 Understanding Existing Solutions

This dissertation aimed to advance the state of the art in JSON-compatible space-efficient serialization. Therefore, attaining extensive knowledge of the inner workings of the existing solutions was a pre-requisite. However, I found that existing publications do not explore JSON-compatible serialization specifications beyond trivial examples. This was a motivation to write and publish the paper *A Survey of JSON-compatible Binary Serialization Specifications* [151]. Filling that gap in the literature took longer than expected given I underestimated the complexity of getting intimately familiar with 13 binary serialization specifications. However, the knowledge earned through this study was key to the success of the project. By the end of the study, I was proficient enough in binary serialization to contribute fixes and clarifications to 6 popular open-source JSON-compatible binary serialization specifications. The resulting >100 pages paper including annotated bit-strings for every considered serialization specification could not be included in its entirety on this dissertation due to space constrains. However, I provide a summary of the study throughout section 5.1, chapter 6 and chapter 10.

11.2 Measuring Existing Solutions

As explained in section 2.2, I found existing space-efficiency JSON-compatible benchmark literature to be insufficient to draw comprehensive conclusions. In particular, the input data considered by existing benchmarks was not representative and it seems was collected without following any methodical selection process. This led to the motivation to write and publish the paper *A Benchmark of JSON-compatible Binary Serialization Specifications* [150]. This extensive benchmark study provided key insights about the combination of characteristics, optimizations and trade-offs in binary serialization specifications that result in space-efficiency. The resulting >100 pages publication could not be included in its entirety on this dissertation due to space constrains. However, I provided a summary of such study throughout section 5.2, chapter 4, and chapter 7.

In comparison to existing benchmark literature, the core insight behind my study was a solution to the input data selection process based on the definition of the formal taxonomy presented in section 4.2 to classify JSON documents based on aspects that impact data serialization. Balancing the granularity and the breadth of the formal taxonomy defined in section 4.2 was a challenge as considering additional dimensions exponentially increased the amount of categories defined by the taxonomy. For this study, I wrote 189 schema definitions using 7 schema languages over the course of two weeks. Because writing high-quality schema definitions is an arduous process, adding more categories to the taxonomy would have greatly increased the time needed to complete the study. In terms of schema-driven specifications, is it essential to write comparable schema definitions that take advantage of the features provided by each serialization specification for fairness reasons. I attempted to reduce the risk of producing an unfair benchmark by writing the most-advanced possible schema definition for each input document given the documented features and suggestions. Whilst there is a chance I missed certain optimizations, I embraced this potential problem to argue that documentation is an integral part of a software product and that the resulting benchmark also captures the ability of each specification to explain a software engineer how to use its available features to achieve space-efficiency.

11.3 Designing a Novel Solution

The in-depth knowledge in JSON-compatible binary serialization and space-efficiency serialization techniques acquired on section 11.1 and section 11.2 provided key insights into how to architect a novel JSON-compatible space-efficiency schema-driven binary serialization specification. The most important conclusion drawn from the benchmark study presented in chapter 7 is that no approach for serializing a data type is strictly superior for every case. This design decision was the fundamental principle that influenced the rest of the architecture and forced the conceptual separation between the static analysis phase performed at build-time and the encoding phase performed at runtime. The existing schema-driven serialization specifications studied in chapter 6 often aim to achieve space-efficiency by embracing simplicity in their schema languages. However, I found that the complexity and expressiveness of JSON Schema [159] as discussed in subsubsection 3.2.3.1 was key in enabling advanced space-efficient encodings and optimizations that are not possible with simpler schema languages.

Defining a set of sensible transformation rules for the Canonicalizer component introduced in subsection 8.3.2 was a challenge. The difficulty was in finding the right balance for simplification without accidentally hindering the ability to spot space-efficiency optimizations due to over-simplification. I solved this problem by clearly defining the

objectives of the canonicalization process, outlining a set of guidelines to evaluate the suitability of every potential transformation rule and prioritizing transformation rules that simplified the Mapper component in practice instead of considering hypothetical cases.

11.4 Implementing the Proposed Solution

To ensure development productivity, I integrated the proof-of-concept implementation with the automated benchmark software produced in chapter 7 while on the development phase to get instant feedback for every change to the specification. Such tight feedback loop was the key for rapidly testing space-efficiency optimization experiments and discard ideas that did not produce space-efficient results in practice.

11.4.1 JavaScript Numeric Data Types

In an attempt to extend the reach of JSON BinPack, I selected the TypeScript transpiler to produce an implementation that could run directly on web browsers. However, implementing serialization and deserialization procedures using TypeScript proved to be a challenge as the substandard numeric capabilities of the JavaScript programming language led to subtle precision and bitwise operation problems. For example, evaluating the arithmetic expression $11492746249590654 - 1$ incorrectly results in 11492746249590652. Surprisingly, using the *BigInt* type [1] results in a different failure: $\mathit{BigInt}(11492746249590655) - \mathit{BigInt}(1)$ results in $1149274624959065\mathit{5n}$. Due to the lack of an unsigned integer type, introducing bitwise operators in an arithmetic calculation forces JavaScript interpreters to cast any positive integer that represents a valid negative integer using Two's Complement [151] to a negative integer. For example, $4294967276 << 1$ results in -40. As a consequence, it was difficult to operate on large integer values and get predictable results without forcing the interpreter to cast the value back to a positive integer at every calculation.

11.4.2 JSON Schema Runtime Applicators

JSON Schema applicators are keywords that take JSON Schema definitions as arguments. While many applicator keywords are consumed during the static analysis phase, some applicators must be evaluated at runtime. For this reason, encodings that implement support for runtime applicators must embed a complete JSON Schema implementation to resolve the applicator at runtime. These runtime applicators represent an exception to the architecture as they force the Encoder component introduced in subsection 8.3.3 to understand the concept of JSON Schema instead of remaining agnostic to the schema-language. This weakness of the architecture and potential runtime inefficiency problem can be solved by adopting a code-generation approach to schema refinement where the input JSON Schema definition is converted into specialized source code that serializes and deserializes input data given such schema. With this approach, the implementation generates code that is agnostic to JSON Schema to address the validation problem.

11.4.3 JSON Schema Support

Due to time constrains, it was not possible to define JSON BinPack encoding and mapping rules that cover every keyword defined by JSON Schema within the scope of this dissertation. Instead, I focused on implementing the minimal amount of encoding and mapping rules necessary to prove the space-efficiency characteristics of JSON BinPack in terms of the JSON documents considered by the benchmark study discussed in section 11.2. Instead of imposing restrictions to the JSON Schema definitions that can be utilized with JSON BinPack, I designed the proof-of-concept implementation to enable the incremental support for new JSON Schema keywords by deriving space-efficient encodings from keywords that are understood by the implementation and falling back to a schema-less mode of operation for subschemas that cannot be mapped to more specific encodings.

11.5 Final Conclusions

Overall, the project successfully achieved a proof-of-concept prototype. The time investment in fulfilling the prerequisites resulted in a novel JSON-compatible serialization specification that proved to be strictly space-efficient in comparison to every alternative serialization specifications discussed within the scope of this study. Also, JSON

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt

BinPack resulted in higher average and median space-efficiency improvements with no negative size-reduction cases in comparison to the considered general-purpose data compressors. Furthermore, JSON BinPack in the optional schema-less mode proved to be strictly space-efficient in comparison to every alternative schema-less serialization specification for every JSON document considered in the benchmark study. Additionally, JSON BinPack is able to perform its task without requiring the adoption of custom JSON Schema vocabularies. This characteristic lowers the barrier of adoption by enabling JSON BinPack to be used on any existing software system that adopts JSON Schema.

Research-level Preparation and Upskilling. I spent 6 months of part-time work analysing, understanding and benchmarking existing serialization specifications. I read 250+ papers on the topic. As part of the supervision, I developed research-level skills. With the acquired knowledge, I designed and implemented JSON BinPack over the course of 2 months of part-time work.

Study. This study does not involve human participants and meets the criteria set for legal, ethical, social and professional standards.

12 | Future Work

Based on the research undertaken for this dissertation, I believe there is room for improvement in the following areas:

- **Semantic Schema Versioning.** To the best of my knowledge, there is no human readable versioning scheme that can distingush between backwards and forwards compatible changes. Software libraries typically rely on *Semantic Versioning* [1] to succintly communicate whether a software library update is safe by distingushing between incompatible changes, backwards-compatible new functionality and backwards-compatible bug fixes. I envision a versioning convention that is applicable to schemas and distingushes between backwards, forwards, and fully compatible changes.

- **Increased Benchmark Coverage.** I believe there is room to augment the input data set of the space-efficiency benchmark introduced in chapter 7 to include JSON documents that match the 9 missing taxonomy categories described in subsection 5.2.3 and to increase the sample proportionality.

- **Production-ready JSON BinPack Implementation.** In section 8.4, I present a proof-of-concept implementation of JSON BinPack written using the TypeScript programming language to demonstrate the space-efficiency potential of the proposed serialization specification. Because of the problems with the programming language of choice discussed in subsection 11.4.1, I intend to produce a production-ready implementation of JSON BinPack using a systems programming language.

- **JSON BinPack Code Generation Support.** As discussed in [151], schema-driven serialization specifications tend to support code generation to deliver optimal runtime-efficiency. Furthermore, subsection 11.4.2 presented an architectural weakness of JSON BinPack that is solved by adopting a code-generation approach. Therefore, I intend to introduce a code-generation component to the JSON BinPack architecture.

- **Full JSON Schema Support in JSON BinPack.** As discussed in subsection 11.4.3, the proof-of-concept JSON BinPack implementation introduced in section 8.4 does not support every keyword defined by the JSON Schema Core [159] and Validation [160] specifications. In the interest of time, the proof-of-concept implementation only supports the subset of JSON Schema keywords that were necessary to model the input JSON documents considered by the benchmark study defined in chapter 7. I intend to continue writing encodings, canonicalization and mapping rules to cover every official JSON Schema vocabulary.

[1] https://semver.org

Bibliography

[1] 3GPP. 2021. *Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 Application Protocol (S1AP)*. 3GPP. https://www.3gpp.org/ftp/Specs/archive/36_series/36.413/36413-g40.zip

[2] J. Alakuijala and Z. Szabadka. 2016. *Brotli Compressed Data Format*. RFC. IETF. https://doi.org/10.17487/RFC7932

[3] C. Andrei, D. Florescu, G. Fourny, J. Robie, and P. Velikhov. [n.d.]. *Edition specification version 2.0.8 for JSound 2.0*. http://www.jsound-spec.org/publish/en-US/JSound/2.0/html-single/JSound/index.html#appe-JSound-Revision_History

[4] Edman Anjos, Junhee Lee, and Srinivasa Rao Satti. 2016. SJSON: A succinct representation for JavaScript object notation documents. , 173–178 pages.

[5] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Counting Types for Massive JSON Datasets. In *Proceedings of The 16th International Symposium on Database Programming Languages*. Association for Computing Machinery, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/3122831.3122837

[6] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *The VLDB Journal* 28, 4 (2019), 497–521.

[7] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas and Types for JSON Data: From Theory to Practice. 2060–2063. https://doi.org/10.1145/3299869.3314032

[8] Rahwa Bahta and Mustafa Atay. 2019. Translating JSON Data into Relational Data Using Schema-Oblivious Approaches. In *Proceedings of the 2019 ACM Southeast Conference* (Kennesaw, GA, USA) *(ACM SE '19)*. Association for Computing Machinery, New York, NY, USA, 233–236. https://doi.org/10.1145/3299815.3314467

[9] Guido Barbaglia, Simone Murzilli, and Stefano Cudini. 2017. Definition of REST web services with JSON schema. , 907–920 pages.

[10] Matěj Bartík, Sven Ubik, and Pavel Kubalik. 2015. LZ4 compression algorithm on FPGA. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. 179–182. https://doi.org/10.1109/ICECS.2015.7440278

[11] Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.

[12] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. 2009. *YAML Ain't Markup Language (YAML) Version 1.2*. YAML. https://yaml.org/spec/1.2/spec.html

[13] Clara Benac Earle, Lars-Åke Fredlund, Ángel Herranz, and Julio Mariño. 2014. Jsongen: A Quickcheck Based Library for Testing JSON Web Services. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang* (Gothenburg, Sweden) *(Erlang '14)*. Association for Computing Machinery, New York, NY, USA, 33–41. https://doi.org/10.1145/2633448.2633454

[14] Amrit Kumar Biswal and Obada Almallah. 2019. *Analytical Assessment of Binary Data Serialization Techniques in IoT Context*. Master's thesis. Dipartimento di Elettronica, Informazione e Bioingegneria.

[15] Sebastian Bittl, Arturo Gonzalez, and Wolf Heidrich. 2014. Performance comparison of encoding schemes for ETSI ITS C2X communication systems.

[16] Daniele Bonetta and Matthias Brantner. 2017. FAD.Js: Fast JSON Data Access Using JIT-Based Speculative Optimizations. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1778–1789. https://doi.org/10.14778/3137765.3137782

[17] C. Bormann and P. Hoffman. 2013. *Concise Binary Object Representation (CBOR)*. RFC. IETF. https://doi.org/10.17487/RFC7049

[18] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. 2017. JSON: Data Model, Query Languages and Schema Specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/3034786.3056120

[19] Anthony R. Bradley, Alexander S. Rose, Antonín Pavelka, Yana Valasatava, Jose M. Duarte, Andreas Prlić, and Peter W. Rose. 2017. MMTF—An efficient file format for the transmission, visualization, and analysis of macromolecular structures. *PLOS Computational Biology* 13, 6 (06 2017), 1–16. https://doi.org/10.1371/journal.pcbi.1005575

[20] Steven Braeger. 2016. *Universal Binary JSON Specification*. UBJSON. https://ubjson.org

[21] T. Bray. 2014. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC. IETF. https://doi.org/10.17487/RFC8259

[22] Tim Bray and C. M. Sperberg-McQueen. 1996. *Extensible Markup Language (XML)*. W3C Working Draft. W3C. https://www.w3.org/TR/WD-xml-961114.

[23] Simon Brown. [n.d.]. *The C4 model for visualising software architecture*. https://c4model.com

[24] K. Brun-Laguna, T. Watteyne, S. Malek, Z. Zhang, C. Oroza, S. D. Glaser, and B. Kerkez. 2016. SOL: An end-to-end solution for real-world remote monitoring systems. In *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. IEEE, Valencia, Spain, 1–6. https://doi.org/10.1109/PIMRC.2016.7794864

[25] P. Bryan. 2013. *JavaScript Object Notation (JSON) Patch*. RFC. IETF. https://doi.org/10.17487/RFC6902

[26] P. Bryan. 2013. *JavaScript Object Notation (JSON) Pointer*. RFC. https://doi.org/10.17487/RFC6901

[27] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, and T. Schaub. 2016. *The GeoJSON Format*. RFC. IETF. https://doi.org/10.17487/RFC7946

[28] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2013. Discovering Implicit Schemas in JSON Data. In *Web Engineering*, Florian Daniel, Peter Dolog, and Qing Li (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–83.

[29] Hanyang Cao, Jean-Rémy Falleri, Xavier Blanc, and Li Zhang. 2016. JSON Patch for Turning a Pull REST API into a Push. In *Service-Oriented Computing*, Quan Z. Sheng, Eleni Stroulia, Samir Tata, and Sami Bhiri (Eds.). Springer International Publishing, Cham, 435–449.

[30] U. Carion. 2020. *JSON Type Definition*. Technical Report. IETF. draft-ucarion-json-type-definition-04

[31] David Carrera, Jonathan Rosales, and Gustavo A. 2018. Optimizing Binary Serialization with an Independent Data Definition Format. *International Journal of Computer Applications* 180 (03 2018), 15–18. https://doi.org/10.5120/ijca2018916670

[32] CCSDS. 2016. *Space Link Extension — Return Channel Frames Service Specification*. CCSDS. https://public.ccsds.org/Pubs/911x2b3.pdf

[33] V.G. Cerf. 1969. *ASCII format for network interchange*. STD. IETF. https://doi.org/10.17487/RFC0020

[34] Alberto Hernandez Chillon, Diego Sevilla Ruiz, and Jesus Garcia Molina. 2020. Deimos: a model-based NoSQL data generation language. In *International Conference on Conceptual Modeling*. Springer, Springer, Vienna, Austria, 151–161.

[35] Koen Claessen and John Hughes. 2011. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 46, 4 (May 2011), 53–64. https://doi.org/10.1145/1988042.1988046

[36] James Clark and MURATA Makot. 2001. *RELAX NG Specification*. Oasis Open. https://www.oasis-open.org/committees/relax-ng/spec-20011203.html

[37] Yann Collet and Murray Kucherawy. 2021. *Zstandard Compression and the 'application/zstd' Media Type*. RFC. IETF. https://doi.org/10.17487/RFC8878

[38] The Unicode Consortium. 2019. *The Unicode Standard, Version 12.1.0*. Standard. The Unicode Consortium, Mountain View, CA.

[39] Osborne Computer Corporation. 1983. *Osborne EXECUTIVE Reference Guide*. Osborne Computer Corporation, San Francisco, CA.

[40] Douglas Crockford. 2006. *JSON*. IETF. https://tools.ietf.org/html/draft-crockford-jsonorg-json-00

[41] Douglas Crockford. 2011. *Douglas Crockford: The JSON Saga*. Youtube. https://www.youtube.com/watch?v=-C-JoyNuQJs

[42] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2016. JSONDiscoverer: Visualizing the schema lurking behind JSON documents. *Knowledge-Based Systems* 103 (2016), 52–55. https://doi.org/10.1016/j.knosys.2016.03.020

[43] ISO/IEC JTC 1/SC 34 Document description and processing languages. 1986. *Standard Generalized Markup Language (SGML)*. Standard. International Organization for Standardization.

[44] P. Deutsch. 1996. *GZIP file format specification version 4.3*. RFC. https://doi.org/10.17487/RFC1952

[45] Olivier Dubuisson and Philippe Fouquart. 2001. *ASN.1: Communication between Heterogeneous Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[46] ECMA. 2017. *ECMA-404: The JSON Data Interchange Syntax*. ECMA, Geneva, CH. https://www.ecma-international.org/publications/standards/Ecma-404.htm

[47] ECMA. 2021. *ECMA-262: ECMAScript 2021 language specification*. ECMA, Geneva, CH. https://www.ecma-international.org/publications/standards/Ecma-262.htm

[48] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2017. Example-Driven Web API Specification Discovery. In *Modelling Foundations and Applications*, Anthony Anjorin and Huáscar Espinoza (Eds.). Springer International Publishing, Cham, 267–284.

[49] Malin Eriksson and Victor Hallberg. 2011. Comparison between JSON and YAML for data serialization. *The School of Computer Science and Engineering Royal Institute of Technology* 0, 0 (2011), 1–25.

[50] Paola Espinoza-Arias, Daniel Garijo, and Oscar Corcho. 2020. Mapping the Web Ontology Language to the OpenAPI Specification. In *International Conference on Conceptual Modeling*. Springer, Springer, Vienna, Austria, 117–127.

[51] G. Court F. Galiegue, K. Zyp. [n.d.]. *JSON Schema: core definitions and terminology*. IETF. https://tools.ietf.org/html/draft-zyp-json-schema-04

[52] Steve Faulkner, Arron Eicholz, Travis Leithead, Alex Danilo, and Sangwhan Moon. 2021. *HTML 5.2*. W3C Recommendation. W3C. https://www.w3.org/TR/html52/.

[53] Jianhua Feng and Jinhong Li. 2013. Google protocol buffers research and application in online game. In *IEEE Conference Anthology*. IEEE, China, 4.

[54] Paolo Ferragina and Giovanni Manzini. 2010. On Compressing the Textual Web. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining* (New York, New York, USA) *(WSDM '10)*. Association for Computing Machinery, New York, NY, USA, 391–400. https://doi.org/10.1145/1718487.1718536

[55] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999. *Hypertext Transfer Protocol – HTTP/1.1*. RFC. https://doi.org/10.17487/RFC2616

[56] R. Fielding and J. Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC. https://doi.org/10.17487/RFC7231

[57] George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes* 22, 4 (1997), 74–80.

[58] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-Specific Language for Processing Ad Hoc Data. *SIGPLAN Not.* 40, 6 (June 2005), 295–304. https://doi.org/10.1145/1064978.1065046

[59] Ivan Flores. 1963. *The Logic Of Computer Arithmetic*. Prentice-Hall, Newport Coast, CA, USA, 24.

[60] Working Group for Floating-Point Arithmetic. 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* 1, 1 (2019), 1–84.

[61] The Apache Software Foundation. 2012. *Apache Avro™ 1.10.0 Specification*. The Apache Software Foundation. https://avro.apache.org/docs/current/spec.html

[62] A. A. Frozza, R. d. S. Mello, and F. d. S. d. Costa. 2018. An Approach for Schema Extraction of JSON and Extended JSON Document Collections. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, Salt Lake City, Utah, 356–363.

[63] Michael Fruth, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas. In *International Conference on Conceptual Modeling*. Springer, Springer, Vienna, Austria, 220–230.

[64] Sadayuki Furuhashi, Satoshi Tagomori, Yuichi Tanikawa, Stephen Colebourne, Stefan Friesel, René Kijewski, Michael Cooper, Uenishi Kota, and Gabe Appleton. 2013. *MessagePack Specification*. MessagePack. https://github.com/msgpack/msgpack/blob/master/spec.md

[65] Shudi Gao, C. M. Sperberg-McQueen, Henry S. Thompson, Noah Mendelsohn, David Beech, and Murray Maloney. 2007. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C Working Group Note. W3C. https://www.w3.org/TR/2007/WD-xmlschema11-1-20070830/.

[66] Bruno Gil and Paulo Trezentos. 2011. Impacts of Data Interchange Formats on Energy Consumption and Performance in Smartphones. In *Proceedings of the 2011 Workshop on Open Source and Design of Communication* (Lisboa, Portugal) *(OSDOC '11)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2016716.2016718

[67] Google. 2020. *Protocol Buffers Version 3 Language Specification*. Google. https://developers.google.com/protocol-buffers/docs/reference/proto3-spec

[68] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2019. Type Safety with JSON Subschema. arXiv:1911.12651 [cs.PL]

[69] P. Hallam-Baker. 2018. *Binary Encodings for JavaScript Object Notation: JSON-B, JSON-C, JSON-D*. Technical Report. IETF. https://tools.ietf.org/html/draft-hallambaker-jsonbcd-11#section-6

[70] Jean Carlo Hamerski, Anderson RP Domingues, Fernando G Moraes, and Alexandre Amory. 2018. Evaluating serialization for a publish-subscribe based middleware for MPSoCs. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, IEEE, Bordeaux, 773–776.

[71] Sam Harrison, Abhishek Dasgupta, Simon Waldman, Alex Henderson, and Christopher Lovell. 2021. How reproducible should research software be? https://doi.org/10.5281/zenodo.4761867

[72] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2012. Edit lenses. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 495–508.

[73] R Nigel Horspool and Gordon V Cormack. 1992. Constructing Word-Based Text Compression Algorithms.. In *Data Compression Conference*. 62–71.

[74] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[75] IBM. 1972. *IBM FORTRAN Program Products for OS and the CMS Component of VM/370 General Information*. IBM. http://bitsavers.trailing-edge.com/pdf/ibm/370/fortran/GC28-6884-0_IBM_FORTRAN_Program_Products_for_OS_and_CMS_General_Information_Jul72.pdf

[76] IBM. 1986. *Code Page CPGID 00037*. IBM. ftp://ftp.software.ibm.com/software/globalization/gcoc/attachments/CP00037.pdf

[77] Pietro Incardona, Antonio Leo, Yaroslav Zaluzhnyi, Rajesh Ramaswamy, and Ivo F Sbalzarini. 2019. OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers. *Computer Physics Communications* 241 (2019), 155–177.

[78] their environments ISO/IEC JTC 1/SC 22 Programming languages and system software interfaces. 2002. *Z formal specification notation — Syntax, type system and semantics*. Standard. International Organization for Standardization.

[79] their environments ISO/IEC JTC 1/SC 22 Programming languages and system software interfaces. 2011. *Information technology — Programming languages — C++*. Standard. International Organization for Standardization.

[80] data elements ISO/TC 154, Processes, industry documents in commerce, and administration. 2004. *Data elements and interchange formats – Information interchange – Representation of dates and times*. Standard. International Organization for Standardization, Geneva, CH.

[81] Massimiliano Izzo. 2016. *The JSON-Based Data Model*. Springer International Publishing, Cham, 39–48. https://doi.org/10.1007/978-3-319-31241-5_3

[82] K. Jahed and J. Dingel. 2019. Enabling Model-Driven Software Development Tools for the Internet of Things. In *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*. IEEE, Montreal, QC, Canada, 93–99. https://doi.org/10.1109/MiSE.2019.00022

[83] Uthayakumar Jayasankar, Vengattaraman Thirumal, and Dhavachelvan Ponnurangam. 2021. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University - Computer and Information Sciences* 33, 2 (2021), 119–140. https://doi.org/10.1016/j.jksuci.2018.05.006

[84] Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2015. Schema extraction and structural outlier detection for JSON-based nosql data stores. In *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.). Gesellschaft für Informatik e.V., Bonn, 425–444.

[85] G. Klyne and C. Newman. 2002. *Date and Time on the Internet: Timestamps*. RFC. IETF. https://doi.org/10.17487/RFC3339

[86] Ronny Krashinsky. 2003. Efficient web browsing for mobile clients using HTTP compression. *See: http://www. cag. lcs. mit. edu/~ ronny/classes/httpcomp. pdf* (2003).

[87] Pavel Kyurkchiev. 2015. Integrating a System for Symbol Programming of Real Processes with a Cloud Service. , 8 pages.

[88] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. , 941–960 pages.

[89] John Larmouth. 1999. *ASN.1 Complete*. Open Systems Solutions, 1748 Millstream Way, Henderson, NV 89014, Chapter 4.2 LWER - Light-Weight Encoding Rules, 316–319.

[90] H. Ledoux, G.A.K. Arroyo Ohori, K. Kavisha, B. Dukai, A. Labetski, and S. Vitalis. 2019. CityJSON: a compact and easy-to-use encoding of the CityGML data model. , urn:issn:2363–7501 pages.

[91] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *Proc. VLDB Endow.* 10, 10 (June 2017), 1118–1129. https://doi.org/10.14778/3115404.3115416

[92] K Maeda. 2012. Performance evaluation of object serialization libraries in XML, JSON and binary formats. , 177–182 pages.

[93] Benjamin Maiwald, Benjamin Riedle, and Stefanie Scherzinger. 2019. What Are Real JSON Schemas Like?. In *Advances in Conceptual Modeling*, Giancarlo Guizzardi, Frederik Gailly, and Rita Suzana Pitangueira Maciel (Eds.). Springer International Publishing, Cham, 95–105.

[94] Simon Marlow et al. 2010. Haskell 2010 language report. *Available on: https://www. haskell. org/onlinereport/haskell2010* (2010).

[95] Dwight Merriman, Stephen Steneker, Hannes Magnusson, Luke Lovett, Kevin Albertson, Kay Kim, and Allison Reinheimer Moore. 2020. *BSON Specification Version 1.1*. MongoDB. http://bsonspec.org/spec.html

[96] Microsoft. 2018. *Bond Compiler 0.12.0.1*. Microsoft. https://microsoft.github.io/bond/manual/compiler.html

[97] N. Mitra. 1994. Efficient encoding rules for ASN.1-based protocols. *AT T Technical Journal* 73, 3 (1994), 80–93. https://doi.org/10.1002/j.1538-7305.1994.tb00590.x

[98] Muhammad Wasim Bhatti, F. Hayat, N. Ehsan, A. Ishaque, S. Ahmed, and E. Mirza. 2010. A methodology to manage the changing requirements of a software project. In *2010 International Conference on Computer Information Systems and Industrial Management Applications (CISIM)*. IEEE, Krakow, Poland, 319–322. https://doi.org/10.1109/CISIM.2010.5643642

[99] Gerald Neufeld and Son Vuong. 1992. An overview of ASN.1. *Computer Networks and ISDN Systems* 23, 5 (1992), 393–415. https://doi.org/10.1016/0169-7552(92)90014-H

[100] Mattias Nordahl and Boris Magnusson. 2015. A lightweight Data Interchange Format for Internet of Things in the PalCom Middleware Framework. , 284–291 pages.

[101] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. 2009. Comparison of JSON and XML data interchange formats: a case study. *Caine* 9 (2009), 157–162.

[102] L. Opyrchal and A. Prakash. 1999. Efficient object serialization in Java. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware*. IEEE, Austin, TX, USA, 96–101. https://doi.org/10.1109/ECMDD.1999.776421

[103] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter before You Parse: Faster Analytics on Raw Data with Sparser. *Proc. VLDB Endow.* 11, 11 (July 2018), 1576–1589. https://doi.org/10.14778/3236187.3236207

[104] Jean Paoli, François Yergeau, Tim Bray, Eve Maler, and Michael Sperberg-McQueen. 2006. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C Recommendation. W3C. https://www.w3.org/TR/2006/REC-xml-20060816/.

[105] Akshay Parashar, Payal Anand, and Arun Abraham. 2020. Performance Analysis and Optimization of Serialization Techniques for Deep Neural Networks. In *Computer Vision, Pattern Recognition, Image Processing, and Graphics*, R. Venkatesh Babu, Mahadeva Prasanna, and Vinay P. Namboodiri (Eds.). Springer Singapore, Singapore, 250–260.

[106] PM Parekar and SS Thakare. 2014. Lossless data compression algorithm–a review. *International Journal of Computer Science & Information Technologies* 5, 1 (2014).

[107] C. Peng, P. Goswami, and G. Bai. 2018. Fuzzy Matching of OpenAPI Described REST Services. *Procedia Computer Science* 126, 1313–1322.

[108] Roger D Peng. 2011. Reproducible research in computational science. *Science* 334, 6060 (2011), 1226–1227.

[109] Bo Petersen, Henrik Bindner, Shi You, and Bjarne Poulsen. 2017. Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the internet of things. In *2017 Computing Conference*. IEEE, IEEE, London, UK, 1339–1346.

[110] Dušan Petković. 2020. Non-Native Techniques for Storing JSON Documents into Relational Tables. In *Proceedings of the 22nd International Conference on Information Integration and Web-Based Applications & Services* (Chiang Mai, Thailand) *(iiWAS '20)*. Association for Computing Machinery, New York, NY, USA, 16–20. https://doi.org/10.1145/3428757.3429103

[111] F. Pezoa, J.L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. 2016. Foundations of JSON schema. , 263–273 pages.

[112] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić. 2016. Performance evaluation of using Protocol Buffers in the Internet of Things communication. In *2016 International Conference on Smart Systems and Technologies (SST)*. International Conference on Smart Systems and Technologies, Osijek, HR, 261–265.

[113] M. A. Pradana, A. Rakhmatsyah, and A. A. Wardana. 2019. Flatbuffers Implementation on MQTT Publish/Subscribe Communication as Data Delivery Format. In *2019 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. IEEE, Bandung, Indonesia, 142–146. https://doi.org/10.23919/EECSI48112.2019.8977050

[114] Tom Preston-Werner and Pradyun Gedam. 2020. *TOML v1.0.0-rc.2*. TOML. https://toml.io/en/v1.0.0-rc.2

[115] D. P. Proos and N. Carlsson. 2020. Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV. In *2020 IFIP Networking Conference (Networking)*. IEEE, Paris, France, 10–18.

[116] Hilary Putnam. 1957. Three-valued logic. *Philosophical Studies* 8, 5 (1957), 73–80.

[117] Ricardo Queirós. 2014. JSON on Mobile: is there an Efficient Parserl. In *3rd Symposium on Languages, Applications and Technologies (OpenAccess Series in Informatics (OASIcs), Vol. 38)*, Maria Jo ao Varanda Pereira, José Paulo Leal, and Alberto Sim oes (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 93–100. https://doi.org/10.4230/OASIcs.SLATE.2014.93

[118] John F. Roddick. 1995. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology* 37 (1995), 383–393.

[119] Kristina Sahlmann, Alexander Lindemann, and Bettina Schnor. 2018. Binary Representation of Device Descriptions: CBOR versus RDF HDT. *Technische Universität Braunschw: Braunschweig, Germany* 0, 0 (2018), 4.

[120] Y. Sakamoto, S. Matsumoto, S. Tokunaga, S. Saiki, and M. Nakamura. 2015. Empirical study on effects of script minification and HTTP compression for traffic reduction. In *2015 Third International Conference on Digital Information, Networking, and Wireless Communications (DINWC)*. 127–132. https://doi.org/10.1109/DINWC.2015.7054230

[121] Tatu Saloranta. 2017. *Efficient JSON-compatible binary format: "Smile"*. FasterXML. https://github.com/FasterXML/smile-format-specification/blob/master/smile-specification.md

[122] ITU Telecommunication Standardization Sector. 1984. *Message handling systems: presentation transfer syntax and notation*. Standard. ITU Telecommunication Standardization Sector.

[123] ITU Telecommunication Standardization Sector. 2015. *Abstract Syntax Notation One (ASN.1): Specification of basic notation*. Standard. ITU Telecommunication Standardization Sector.

[124] ITU Telecommunication Standardization Sector. 2015. *ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*. Standard. ITU Telecommunication Standardization Sector.

[125] ITU Telecommunication Standardization Sector. 2021. *ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. Standard. ITU Telecommunication Standardization Sector.

[126] David Sferruzza. 2018. Top-down Model-Driven Engineering of Web Services from Extended OpenAPI Models. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 940–943. https://doi.org/10.1145/3238147.3241536

[127] Y. Shafranovich. 2005. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC. IETF. https://doi.org/10.17487/RFC4180

[128] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. 2000. Fast and Flexible Word Searching on Compressed Text. *ACM Trans. Inf. Syst.* 18, 2 (April 2000), 113–139. https://doi.org/10.1145/348751.348754

[129] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5 (2007), 8.

[130] Simon Stewart and David Burns. 2020. *WebDriver*. W3C Working Draft. W3C.

[131] James A. Storer and Thomas G. Szymanski. 1982. Data Compression via Textual Substitution. *J. ACM* 29, 4 (Oct. 1982), 928–951. https://doi.org/10.1145/322344.322346

[132] API Storytelling. 2021. *API Storytelling with Ben Hutton*. Youtube. https://www.youtube.com/watch?v=4xbA82lo_lc

[133] Audie Sumaray and S Makki. 2012. A comparison of data serialization formats for optimal efficiency on a mobile platform. , 6 pages.

[134] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[135] Pavel Conto˘s˘ and Martin Svoboda. 2020. JSON Schema Inference Approaches. In *Advances in Conceptual Modeling: ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3–6, 2020, Proceedings*. Springer Nature, Springer, Vienna, Austria, 173.

[136] ISO/IEC JTC 1/SC 6 Telecommunications and information exchange between systems. 2014. *ASN.1 encoding rules — Part 7: Specification of Octet Encoding Rules (OER)*. Standard. International Organization for Standardization, Geneva, CH.

[137] ISO/IEC JTC 1/SC 6 Telecommunications and information exchange between systems. 2015. *Abstract Syntax Notation One (ASN.1): Specification of basic notation*. Standard. International Organization for Standardization, Geneva, CH.

[138] ISO/IEC JTC 1/SC 6 Telecommunications and information exchange between systems. 2015. *ASN.1 encoding rules — Part 4: XML Encoding Rules (XER)*. Standard. International Organization for Standardization, Geneva, CH.

[139] ISO/IEC JTC 1/SC 6 Telecommunications and information exchange between systems. 2018. *ASN.1 encoding rules — Part 8: Specification of JavaScript Object Notation Encoding Rules (JER)*. Standard. International Organization for Standardization, Geneva, CH.

[140] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. 2018. Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 413–429.

[141] Jordi van Liempt. 2020. *CityJSON: does (file) size matter?* Master's thesis. Department of Urbanism, Faculty of the Built Environment & Architecture.

[142] Marcel van Lohuizen, Jonathan Amsterdam, Roger Peppe, Daniel Martí, Axel Wagner, and Ivan Diao. [n.d.]. *The CUE Language Specification*. https://cuelang.org/docs/references/spec/

[143] Wouter van Oortmerssen. 2014. *FlatBuffers: Writing a Schema*. Google. https://google.github.io/flatbuffers/flatbuffers_guide_writing_schema.html

[144] Wouter van Oortmerssen. 2017. *FlexBuffers*. Google. https://google.github.io/flatbuffers/flexbuffers.html

[145] J. Vanura and P. Kriz. 2018. Perfomance evaluation of Java, JavaScript and PHP serialization libraries for XML, JSON and binary formats. , 166–175 pages.

[146] Kenton Varda. 2013. *Cap'n Proto Schema Language*. Sandstorm. https://capnproto.org/language.html

[147] Santiago Vargas, Utkarsh Goel, Moritz Steiner, and Aruna Balasubramanian. 2019. Characterizing JSON Traffic Patterns on a CDN. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) *(IMC '19)*. Association for Computing Machinery, New York, NY, USA, 195–201. https://doi.org/10.1145/3355369.3355594

[148] Bill Venners. 2000. *Inside the Java 2 Virtual Machine*. Computing McGraw-Hill.

[149] Juan Cruz Viotti. 2022. *jviotti/binary-json-size-benchmark:*. https://doi.org/10.5281/zenodo.5829569

[150] Juan Cruz Viotti and Mital Kinderkhedia. 2022. A Benchmark of JSON-compatible Binary Serialization Specifications. arXiv:2201.03051 [cs.SE]

[151] Juan Cruz Viotti and Mital Kinderkhedia. 2022. A Survey of JSON-compatible Binary Serialization Specifications. arXiv:2201.02089 [cs.DB]

[152] Kunal Waghray. 2020. JSON Schema Matching: Empirical Observations. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2887–2889. https://doi.org/10.1145/3318464.3384417

[153] P. Wehner, C. Piberger, and D. Göhringer. 2014. Using JSON to manage communication between services in the Internet of Things. In *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, Montpellier, 1–4.

[154] M. Westhead, T. Wen, and R. Carroll. 2003. Describing data on the grid. In *Proceedings. First Latin American Web Congress*. IEEE, Phoenix, AZ, USA, 134–140. https://doi.org/10.1109/GRID.2003.1261708

[155] Canggih Wibowo. 2011. Evaluation of Protocol Buffers as Data Serialization Format for Microblogging Communication.

[156] Jason A Wilkins and Jaakko Järvi. 2016. drys: A Version Control System for Rapid Iteration of Serialization Protocols.

[157] Martin Wischenbart, Stefan Mitsch, Elisabeth Kapsammer, Angelika Kusel, Birgit Pröll, Werner Retschitzegger, Wieland Schwinger, Johannes Schönböck, Manuel Wimmer, and Stephan Lechner. 2012. User Profile Integration Made Easy: Model-Driven Extraction and Transformation of Social Network Schemas. In *Proceedings of the 21st International Conference on World Wide Web* (Lyon, France) *(WWW '12 Companion)*. Association for Computing Machinery, New York, NY, USA, 939–948. https://doi.org/10.1145/2187980.2188227

[158] A. Wright, H. Andrews, and B. Hutton. 2019. *JSON Schema: A Media Type for Describing JSON Documents*. Technical Report. IETF. https://tools.ietf.org/html/draft-handrews-json-schema-02

[159] A. Wright, H. Andrews, and B. Hutton. 2020. *JSON Schema: A Media Type for Describing JSON Documents*. Technical Report. IETF. https://tools.ietf.org/html/draft-bhutton-json-schema-00

[160] A. Wright, H. Andrews, and B. Hutton. 2021. *JSON Schema: A Media Type for Describing JSON Documents*. Technical Report. IETF. https://tools.ietf.org/html/draft-bhutton-json-schema-validation-00

[161] Kamir Yusof and Mustafa Man. 2017. Efficiency of JSON for data retrieval in big data. *Indonesian Journal of Electrical Engineering and Computer Science* 7 (07 2017), 250–262. https://doi.org/10.11591/ijeecs.v7.i1.pp250-262

[162] Yaroslav Zaluzhnyi. 2016. *Serialization and deserialization of complex data structures, and applications in high performance computing*. Ph.D. Dissertation. Technische Universität Dresden.

[163] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

A | Summary of JSON BinPack Encodings

The 34 parameterized encodings defined and implemented for the proof-of-concept JSON BinPack implementation are summarized in Table A.1, Table A.2, Table A.3, Table A.4, Table A.5, Table A.6, Table A.7, Table A.8 and Table A.9. In these definitions, *JSON* corresponds to the set of valid JSON [46] documents, *String* corresponds to every valid sequence of Unicode [38] code-points as defined in [46], *Schema* corresponds to the set of valid JSON Schema 2020-12 [159] definitions and *Encoding* corresponds to the set of valid Encoding Schemas as defined in subsection 8.3.4. The serialization and deserialization implementations for every defined encoding can be found on GitHub [1] and serialization examples can be found in the project documentation [2]. I propose 4 to 10 encodings for each supported data type where each variation is optimized for a difference scenario. Interesting aspects of the proposed set of space-efficient encodings include:

- **Packed Object Properties.** Similar to ASN.1 PER Unaligned [124], the JSON object [46] encodings summarized in Table A.6 and Table A.7 group boolean object values using bit-set data structures where each boolean value occupies 1 bit. In comparison, other serialization specification studied in chapter 6 encode boolean object values using at least 1 byte. Similarly, some object encodings summarized in Table A.7 pack bounded integer values in a bit-aligned manner. For example, a bounded integer type with a range of 7 values is encoded using 3-bits.
- **Custom Real Number Encoding.** Instead of adopting the IEEE 764 [60] floating-point encoding, I propose a space-efficient fixed-point arbitrary-precision real number encoding summarized in Table A.1. This encoding is space-efficient for real numbers that do not consist of a large amount of digits.
- **Shared Strings.** The string encodings that rely on UTF-8 [38] summarized in Table A.3, Table A.4 and Table A.11 are capable of de-duplicating multiple occurrences of the same string values and object keys in a JSON [46] document by encoding pointers to the previous occurrence of the string or to the closest pointer that eventually points to the desired string for data-locality purposes.
- **Word-based Text Compressor.** In Table A.4, I define a space-efficient encoding for serializing text prose based on an alphabet that represents common words as explained in [73]. The input alphabet is configurable to accomodate for different scenarios and text written in different languages.
- **Integer Multiplers.** The integer encodings summarized in Table A.1 are able to increase space-efficiency of encoding bounded integers by the presence of a multiplier that decreases the range of possible values. When the multiplier value is not the unit of multiplication, the range of possible values is encoded as an enumeration.
- **Amortized String Length Prefixes in Schema-less Mode.** In Table A.11, I define 7 string encodings specialized for encoding strings of different lengths to reduce the size of string length markers to a minimum. Each encoding targets an expotentially-growing range of lengths ranging from 0 to 2^{10} UTF-8 [38] code-points.
- **Low-overhead Composite Structures.** The JSON [46] object and array encodings summarized Table A.6, Table A.7 and Table A.5 add minimal structural space overhead to the resulting bit-strings. For example, serializing an object with a single required property or a fixed-length array of one element is equal to serializing the standalone value.
- **Top-level Enumerations.** In Table A.2, I define an enumeration encoding that is applicable to enumerations that are not nested within other structures. For space-efficiency, the first element of the enumeration is represented by not encoding any information on the bit-string.

[1] https://github.com/jviotti/jsonbinpack/tree/c029870287bedfa54afe7b4b5d39e2e95ecff91f/lib/encoder

[2] https://github.com/jviotti/jsonbinpack/tree/c029870287bedfa54afe7b4b5d39e2e95ecff91f/docs/encoder

Table A.1: A high-level overview of the 6 numeric encodings defined for the proof-of-concept JSON BinPack implementation for the Encoder component introduced in subsection 8.3.3.

Encoding Name and Parameters	Description
`BOUNDED_MULTIPLE_8BITS_ENUM_FIXED` *value* : $\mathbb{Z}$ *maximum* : $\mathbb{Z}$ *minimum* : $\mathbb{Z}$ *multiplier* : $\mathbb{N}$	*value* divided by *multiplier*, minus the ceil of *minimum* divided by *multiplier*, encoded as an 8-bit fixed-length unsigned integer.
`FLOOR_MULTIPLE_ENUM_VARINT` *value* : $\mathbb{Z}$ *minimum* : $\mathbb{Z}$ *multiplier* : $\mathbb{N}$	*value* divided by *multiplier*, minus the ceil of *minimum* divided by *multiplier*, encoded as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer.
`ROOF_MULTIPLE_MIRROR_ENUM_VARINT` *value* : $\mathbb{Z}$ *maximum* : $\mathbb{Z}$ *multiplier* : $\mathbb{N}$	The floor of *maximum* divided by *multiplier*, minus *value* divided by *multiplier*, encoded as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer.
`ARBITRARY_MULTIPLE_ZIGZAG_VARINT` *value* : $\mathbb{Z}$ *multiplier* : $\mathbb{N}$	*value* divided by *multiplier* encoded as a ZigZag-encoded Little Endian Base 128 (LEB) [151] variable-length unsigned integer.
`DOUBLE_VARINT_TUPLE` *value* : $\mathbb{R}$	A sequence of two Little Endian ZigZag-encoded Base 128 variable-length [151] unsigned integers: The integer that results from concatenating the integral part and the decimal part of *value* and the position of the decimal mark from the first digit of *value* where a negative integer results in left zero-padding.

Table A.2: A high-level overview of the 4 enumeration encodings defined for the proof-of-concept JSON BinPack implementation for the Encoder component introduced in subsection 8.3.3.

Encoding Name and Parameters	Description
`BOUNDED_CHOICE_INDEX` *value* : *JSON* *choices* : seq *JSON*	The index of *value* in the 8-bit *choices* enumeration encoded as an 8-bit fixed-length unsigned integer.
`LARGE_BOUNDED_CHOICE_INDEX` *value* : *JSON* *choices* : seq *JSON*	The index of *value* in the *choices* enumeration encoded as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer.
`TOP_LEVEL_8BIT_CHOICE_INDEX` *value* : *JSON* *choices* : seq *JSON*	If *value* corresponds to the index 0 to the *choices* enumeration, encode no data. Otherwise, encode the index of *value* in the *choices* enumeration minus 1 as an 8-bit fixed-length unsigned integer.
`CONST_NONE` *value* : *JSON* *choice* : *JSON*	The *value* matches *choice* and is not encoded.

Table A.3: A high-level overview of the first 5 out of 9 string encodings defined for the proof-of-concept JSON BinPack implementation for the Encoder component introduced in subsection 8.3.3. The remaining 4 encodings are listed in Table A.4.

Encoding Name and Parameters	Description
`UTF8_STRING_NO_LENGTH` *value* : *String* *size* : $\mathbb{N}_0$	The fixed-length UTF-8 [38] encoding of the input string.
`FLOOR_PREFIX_LENGTH_ENUM_VARINT` *value* : *String* *minimum* : $\mathbb{N}_0$	The byte-length of *value* minus *minimum* plus 1 as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by the UTF-8 [38] encoding of the value. If *value* has already been encoded, the encoding may consist of the byte constant `0x00` followed by the byte-length of *value* minus *minimum* plus 1 as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer, followed by the current offset minus the offset to the start of *value* in the buffer encoded as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer.
`ROOF_PREFIX_LENGTH_ENUM_VARINT` *value* : *String* *maximum* : $\mathbb{N}_0$	The *maximum* minus the byte-length of *value* plus 1 as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by the UTF-8 [38] encoding of *value*. If *value* has already been encoded, the encoding may consist of the byte constant `0x00` followed by the *maximum* minus the byte-length of *value* plus 1 as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer, followed by the current offset minus the offset to the start of *value* in the buffer encoded as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer.
`BOUNDED_PREFIX_LENGTH_8BIT_FIXED` *value* : *String* *minimum* : $\mathbb{N}_0$ *maximum* : $\mathbb{N}_0$	The byte-length *value* minus *minimum* plus 1 as an 8-bit fixed-length unsigned integer followed by the UTF-8 [38] encoding of *value*. If *value* has already been encoded to the buffer, the encoding may consist of the byte constant `0x00` followed by the byte-length of *value* minus *minimum* plus 1 as an 8-bit fixed-length unsigned integer, followed by the current offset minus the offset to the start of *value* in the buffer encoded as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer.
`STRING_UNBOUNDED_SCOPED_PREFIX_LENGTH` *value* : *String*	The byte-length of *value* plus 1 as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by the UTF-8 [38] encoding of *value*. If *value* has already been encoded to the buffer using this encoding, the encoding is the byte constant `0x00` followed by the current offset minus the offset to the start of the encoding as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer.

Table A.4: This table is a continuation of Table A.3. A high-level overview of the last 4 out of 9 string encodings defined for the proof-of-concept JSON BinPack implementation for the Encoder component introduced in subsection 8.3.3.

Encoding Name and Parameters	Description
`RFC3339_DATE_INTEGER_TRIPLET` *value* : *String*	An RFC3339 [85] date as the sequence of 3 integers: the year as a 16-bit fixed-length Little Endian unsigned integer, the month as an 8-bit fixed-length unsigned integer, and the day as an 8-bit fixed-length unsigned integer.
`URL_PROTOCOL_HOST_REST` *value* : *String*	The sequence of three strings encoded using `FLOOR_PREFIX_LENGTH_ENUM_VARINT` (see Table A.3) with a minimum equal to 0: the protocol excluding the colon, the host excluding the trailing slash, and the rest of the URL including the leading slash.
`STRING_BROTLI` *value* : *String*	The byte-length of *value* as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by *value* compressed using the Brotli lossless compression format [2].
`STRING_DICTIONARY_COMPRESSOR` *value* : *String* *dictionary* : $\mathbb{P}(String \times \mathbb{N}_0)$	The encoding serializes *value* compressing words given the *dictionary*. The encoding starts with the byte-length of *value* as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer. If a word delimited by the ASCII [33] space character is present in the dictionary, the corresponding dictionary entry plus 1 is encoded as a ZigZag-encoded Little Endian Base 128 (LEB) [151] variable-length unsigned integer. The portions of text in between dictionary matches are encoded as the negative string byte-length minus 1 as a ZigZag-encoded Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by the UTF-8 [38] encoding of the string. If the unmatched text portion has already been encoded to the buffer, the encoding may consist of the byte constant `0x00` followed by the byte-length of the string as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer, followed by the current offset minus the offset to the start of the value in the buffer encoded as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer.

Table A.5: A high-level overview of the 5 array encodings defined for the proof-of-concept JSON BinPack implementation for the Encoder component introduced in subsection 8.3.3.

Encoding Name and Parameters	Description
`FIXED_TYPED_ARRAY` *value* : seq *JSON* *size* : $\mathbb{N}_0$ *prefixEncodings* : seq *Encoding* *encoding* : *Encoding*	The elements of the fixed-length *value* array encoded in order. The encoding of the element at a given index is either determined by *prefixEncodings* at the same index or *encoding*.
`FLOOR_TYPED_LENGTH_PREFIX` *value* : seq *JSON* *minimum* : $\mathbb{N}_0$ *prefixEncodings* : seq *Encoding* *encoding* : *Encoding*	The length of *value* minus *minimum* encoded as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by the elements of *value* encoded in order. The encoding of the element at a given index is either determined by *prefixEncodings* at the same index or *encoding*.
`ROOF_TYPED_LENGTH_PREFIX` *value* : seq *JSON* *maximum* : $\mathbb{N}_0$ *prefixEncodings* : seq *Encoding* *encoding* : *Encoding*	*maximum* minus the length of *value* encoded as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by the elements of *value* encoded in order. The encoding of the element at a given index is either determined by *prefixEncodings* at the same index or *encoding*.
`BOUNDED_TYPED_LENGTH_PREFIX` *value* : seq *JSON* *minimum* : $\mathbb{N}_0$ *maximum* : $\mathbb{N}_0$ *prefixEncodings* : seq *Encoding* *encoding* : *Encoding*	If *minimum* equals *maximum*, the elements of *value* encoded in order. Otherwise, the length of *value* minus *minumum* as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by the elements of *value* encoded in order. In both cases, the encoding of the element at a given index is either determined by *prefixEncodings* at the same index or *encoding*.
`BOUNDED_8BITS_TYPED_LENGTH_PREFIX` *value* : seq *JSON* *minimum* : $\mathbb{N}_0$ *maximum* : $\mathbb{N}_0$ *prefixEncodings* : seq *Encoding* *encoding* : *Encoding*	If *minimum* equals *maximum*, the elements of *value* encoded in order. Otherwise, the length of *value* minus *minumum* as a fixed-length 8-bit unsigned integer followed by the elements of *value* encoded in order. In both cases, the encoding of the element at a given index is either determined by *prefixEncodings* at the same index or *encoding*.

Table A.6: A high-level overview of the first 5 out of 10 object encodings defined for the proof-of-concept JSON BinPack implementation for the Encoder component introduced in subsection 8.3.3. The remaining 4 encodings are listed in Table A.7.

Encoding Name and Parameters	Description
`REQUIRED_ONLY_BOUNDED_TYPED_OBJECT` *value* : $\mathbb{P}(String \times JSON)$ *booleanRequiredProperties* : seq *String* *requiredProperties* : seq *String* *propertyEncodings* : $\mathbb{P}(String \times Encoding)$	The boolean properties in *booleanRequiredProperties* encoded in order as a byte-aligned bitset where the least-significant bit corresponds to the first property, followed by the properties in *requiredProperties* encoded in order according to the encodings declared in *propertyEncodings*.
`NON_REQUIRED_BOUNDED_TYPED_OBJECT` *value* : $\mathbb{P}(String \times JSON)$ *optionalProperties* : seq *String* *propertyEncodings* : $\mathbb{P}(String \times Encoding)$	The length of *optionalProperties* as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by a byte-aligned bitset where the least-significant bit corresponds to the first element of *optionalProperties*, followed by the values of *value* encoded in order according to the encodings declared in *propertyEncodings*.
`MIXED_BOUNDED_TYPED_OBJECT` *value* : $\mathbb{P}(String \times JSON)$ *booleanRequiredProperties* : seq *String* *requiredProperties* : seq *String* *optionalProperties* : seq *String* *propertyEncodings* : $\mathbb{P}(String \times Encoding)$	The required properties subset of *value* encoded as `REQUIRED_ONLY_BOUNDED_TYPED_OBJECT` (see Table A.6) followed by the optional properties subset of *value* encoded as `NON_REQUIRED_BOUNDED_TYPED_OBJECT` (see Table A.6).
`ARBITRARY_TYPED_KEYS_OBJECT_WITHOUT_LENGTH` *value* : $\mathbb{P}(String \times JSON)$ *size* : $\mathbb{N}_1$ *encoding* : *Encoding* *keyEncoding* : *Encoding*	Each pair of *value* encoded as the key followed by the value according to *keyEncoding* and *encoding*. The order in which pairs are encoded is undefined.
`ARBITRARY_TYPED_KEYS_OBJECT` *value* : $\mathbb{P}(String \times JSON)$ *encoding* : *Encoding* *keyEncoding* : *Encoding*	The number of key-value pairs in *value* as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by each pair of *value* encoded as the key followed by the value according to *keyEncoding* and *encoding*. The order in which pairs are encoded is undefined.

Table A.7: This table is a continuation of Table A.6. A high-level overview of the last 5 out of 10 object encodings defined for the proof-of-concept JSON BinPack implementation for the Encoder component introduced in subsection 8.3.3.

Encoding Name and Parameters	Description
`REQUIRED_UNBOUNDED_TYPED_OBJECT` *value* : $\mathbb{P}(String \times JSON)$ *booleanRequiredProperties* : seq *String* *requiredProperties* : seq *String* *encoding* : *Encoding* *keyEncoding* : *Encoding* *propertyEncodings* : $\mathbb{P}(String \times Encoding)$	The required properties subset of *value* encoded as `REQUIRED_ONLY_BOUNDED_TYPED_OBJECT` (see Table A.6) followed by the rest of *value* encoded as `ARBITRARY_TYPED_KEYS_OBJECT` (see Table A.6).
`OPTIONAL_UNBOUNDED_TYPED_OBJECT` *value* : $\mathbb{P}(String \times JSON)$ *optionalProperties* : seq *String* *encoding* : *Encoding* *keyEncoding* : *Encoding* *propertyEncodings* : $\mathbb{P}(String \times Encoding)$	The optional properties subset of *value* encoded as `NON_REQUIRED_BOUNDED_TYPED_OBJECT` (see Table A.6) followed by the rest of *value* encoded as `ARBITRARY_TYPED_KEYS_OBJECT` (see Table A.6).
`MIXED_UNBOUNDED_TYPED_OBJECT` *value* : $\mathbb{P}(String \times JSON)$ *booleanRequiredProperties* : seq *String* *requiredProperties* : seq *String* *optionalProperties* : seq *String* *encoding* : *Encoding* *keyEncoding* : *Encoding* *propertyEncodings* : $\mathbb{P}(String \times Encoding)$	The required properties subset of *value* encoded as `REQUIRED_ONLY_BOUNDED_TYPED_OBJECT` (see Table A.6), followed by the optional properties subset of *value* encoded as `NON_REQUIRED_BOUNDED_TYPED_OBJECT` (see Table A.6), followed by the rest of *value* encoded as `ARBITRARY_TYPED_KEYS_OBJECT` (see Table A.6).
`PACKED_UNBOUNDED_OBJECT` *value* : $\mathbb{P}(String \times JSON)$ *booleanRequiredProperties* : seq *String* *requiredProperties* : seq *String* *optionalProperties* : seq *String* *packedRequiredProperties* : seq *String* *packedEncoding* : *Encoding* *encoding* : *Encoding* *keyEncoding* : *Encoding* *propertyEncodings* : $\mathbb{P}(String \times Encoding)$	The length of *packedRequiredProperties* as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by the properties of *value* listed in *packedRequiredProperties* encoded in order followed by the required non-packed properties subset of *value* encoded as `REQUIRED_ONLY_BOUNDED_TYPED_OBJECT` (see Table A.6), followed by the optional non-packed properties subset of *value* encoded as `NON_REQUIRED_BOUNDED_TYPED_OBJECT` (see Table A.6), followed by the rest of *value* encoded as `ARBITRARY_TYPED_KEYS_OBJECT` (see Table A.6). The packed integer properties are encoded as a byte-aligned reversed Little Endian buffer using the least possible amount of bits for each item as determined by the bounds of *packedEncoding*.
`PACKED_BOUNDED_REQUIRED_OBJECT` *value* : $\mathbb{P}(String \times JSON)$ *booleanRequiredProperties* : seq *String* *requiredProperties* : seq *String* *packedRequiredProperties* : seq *String* *packedEncoding* : *Encoding* *propertyEncodings* : $\mathbb{P}(String \times Encoding)$	The properties of *value* listed in *packedRequiredProperties* encoded in order followed by the required non-packed properties subset of *value* encoded as `REQUIRED_ONLY_BOUNDED_TYPED_OBJECT` (see Table A.6). The packed integer properties are encoded as a byte-aligned reversed Little Endian buffer using the least possible amount of bits for each item as determined by the bounds of *packedEncoding*.

Table A.8: A high-level overview of the single union encoding defined for the proof-of-concept JSON BinPack implementation for the Encoder component introduced in subsection 8.3.3.

Encoding Name and Parameters	Description
`CHOICE_INDEX_PREFIX` *value* : *JSON* *schemas* : seq *Schema* *encodings* : seq *Encoding*	The index to the matching choice of *value* in *schemas* as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by *value* encoded as indicated by the corresponding *encodings* entry.

Table A.9: A high-level overview of the single encoding for arbitrary values defined for the proof-of-concept JSON BinPack implementation for the Encoder component introduced in subsection 8.3.3.

Encoding Name and Parameters	Description
`ANY_PACKED_TYPE_TAG_BYTE_PREFIX` *value* : *JSON*	The rules for encoding each supported data type are presented in Table A.10 and Table A.11.

Table A.10: The encoding rules for serializing arbitrary data types using the `ANY_PACKED_TYPE_TAG_BYTE_PREFIX` encoding introduced in Table A.9. This table is continued in Table A.11.

Type	**Condition**	**First 5-bits**	**Next 3-bits**	**Next**
Unsigned Integer	$0 \leq value < 2^5 - 1$	$value + 1$	`101`	Nothing
Unsigned Integer	$2^5 - 1 \leq value < 2^8$	`00000`	`101`	*value* as a fixed-length 8-bit unsigned integer
Unsigned Integer	$value \geq 2^8$	`00011`	`111`	*value* as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer
Signed Integer	$0 > value > -2^5$	$\|value\|$	`110`	Nothing
Signed Integer	$-2^5 \geq value > -2^8$	`00000`	`110`	$\|value\| - 1$ as a fixed-length 8-bit unsigned integer
Signed Integer	$value \leq -2^8$	`00100`	`111`	$\|value\| - 1$ as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer
Real Number	None	`00101`	`111`	`DOUBLE_VARINT_TUPLE` (see Table A.1)
False	None	`00000`	`111`	Nothing
True	None	`00001`	`111`	Nothing
Null	None	`00010`	`111`	Nothing
Object	$\#value < 2^5 - 1$	$\#value + 1$	`011`	Each pair of *value* encoded in arbitrary order as the key followed by the value according to the rules defined in Table A.10 and Table A.11
Object	$\#value \geq 2^5 - 1$	`00000`	`011`	$\#value$ as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by each pair of *value* encoded in arbitrary order as the key followed by the value according to the rules defined in Table A.10 and Table A.11
Array	$\#value < 2^5 - 1$	$\#value + 1$	`100`	The elements of *value* encoded in order using the rules defined in Table A.10 and Table A.11
Array	$\#value \geq 2^5 - 1$	`00000`	`100`	$\#value$ as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by the elements of *value* encoded in order using the rules defined in Table A.10 and Table A.11

Table A.11: Continuation of Table A.10.

Type	Condition	First 5-bits	Next 3-bits	Next
Shared String	$\#value < 2^5 - 1$	$\#value + 1$	`000`	Current offset minus the target offset as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer
Shared String	$\#value \geq 2^5 - 1$	`00000`	`000`	A sequence of two Little Endian Base 128 (LEB) [151] variable-length unsigned integers: $\#value$ and the current offset minus the target offset
String	$\#value < 2^5 - 1$	$\#value + 1$	`001`	*value* as a UTF-8 [38] string
String	$2^5 - 1 \leq \#value < 2^6 - 2$	$\#value - 2^5 - 1$	`010`	*value* as a UTF-8 [38] string
String	$2^6 - 2 \leq \#value < 2^7$	`00000`	`001`	$\#value$ as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by *value* as a UTF-8 [38] string
String	$2^7 \leq \#value < 2^8$	`00111`	`111`	$\#value - 2^7$ as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by *value* as a UTF-8 [38] string
String	$2^8 \leq \#value < 2^9$	`01000`	`111`	$\#value - 2^8$ as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by *value* as a UTF-8 [38] string
String	$2^9 \leq \#value < 2^{10}$	`01001`	`111`	$\#value - 2^9$ as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by *value* as a UTF-8 [38] string
String	$\#value \geq 2^{10}$	`01010`	`111`	$\#value - 2^{10}$ as a Little Endian Base 128 (LEB) [151] variable-length unsigned integer followed by *value* as a UTF-8 [38] string

B | JSON Compatibility Analysis

Implementing the benchmark software and writing schemas for the set of schema-driven serialization specification revealed that some of the considered schema-driven serialization specifications are not strictly compatible with JSON [46]. When utilizing these serialization specifications, some instances of the input data listed in Table 5.3 and Table 5.4 required transformations to be accepted by the implementations or the respective schema definition languages. The required transformations can be inspected in the benchmark public GitHub repository [1]. These transformations are divided into the following categories:

- **Keys.** The schema definition languages provided by ASN.1 [123], Microsoft Bond [96], Cap'n Proto [146], FlatBuffers [143], Protocol Buffers [67], and Apache Thrift [129] disallow property names that include hyphens, slashes, dollar signs, parenthesis, and periods. Also, ASN.1 [123] disallows property names that start with an underscore and Cap'n Proto [146] disallows property names that include underscores and capitalised property names. Furthermore, Protocol Buffers [67] and Apache Thrift [129] disallow property names that equal the reserved keywords *async*, *extends*, *in*, and *with*. To handle these cases, the disallowed properties are renamed to a close variation that the schema language permits.

- **Values.** Protocol Buffers [67] defines the *null* type as an enumeration consisting of a single constant: zero [2]. FlatBuffers [143] does not support a *null* type. When using FlatBuffers [143], I represent this type with an enumeration consisting of a single constant in the same manner as Protocol Buffers [67]. In both cases, I transform any JSON [46] *null* value into zero.

- **Structural.** Neither Microsoft Bond [96], Cap'n Proto [146], FlatBuffers [143], Protocol Buffers [67], nor Apache Thrift [129] support encoding a JSON document that consists of a top level array. In these cases, I move the array into a wrapper structure. FlatBuffers [143] and Protocol Buffers [67] do not support nested arrays. In these cases, I introduce wrapper structures at every array nesting level. ASN.1 [123], Microsoft Bond [96], Cap'n Proto [146], FlatBuffers [143], Protocol Buffers [67], and Apache Thrift [129] do not support heterogenous arrays of non-composite types. In these cases, I convert the heterogenous arrays into arrays of union structures. Microsoft Bond [96] does not support union types. In this case I introduce a structure consisting of optional fields. Additionally, the use of unions in FlatBuffers [143] requires the introduction of an additional textual property to signify the union choice. In order not to put this specification at a disadvantage, I encode the fixed-length heterogenous array as tables where their property names correspond to the array indexes.

The type of transformations that were necessary for each JSON document from the input data defined in Table 5.3 and Table 5.4 are listed in Table B.1. In summary, every schema-less binary serialization specifications listed in Table 5.6 is compatible with the input data set. In terms of schema-driven specifications, only Apache Avro [61] is strictly compatible with the input data set.

[1] https://github.com/jviotti/binary-json-size-benchmark
[2] https://github.com/protocolbuffers/protobuf/blob/master/src/google/protobuf/struct.proto

Table B.1: A summary of the transformations required to serialize the input data JSON documents listed in Table 5.3 and Table 5.4 using the set of binary serialization specifications listed in Table 5.5 and Table 5.6. The JSON documents from Table 5.3 and Table 5.4 that are not present in this table did not require any type of transformation. Each letter signifies the type of required transformation as defined in this section. The letter K stands for *Keys*, the letter V stands for *Values*, and the letter S stands for *Structural*.

Input Data	**ASN.1**	**Apache Avro**	**Microsoft Bond**	**BSON**	**Cap'n Proto**	**CBOR**	**FlatBuffers**	**FlexBuffers**	**MessagePack**	**Protocol Buffers**	**Smile**	**Apache Thrift**	**UBJSON**
Tier 1 NRF	K		K		K		K			K		K	
Tier 1 NRN	K		K		K		K			K		K	
Tier 1 TRF	K		K		K		K			K		K	
Tier 1 TRN	K+S		K+S		K+S		K+S			K+S		K+S	
Tier 1 TNF										K		K	
Tier 1 BRF							V			V			
Tier 1 BRN	K		K		K		K			K		K	
Tier 1 BNN	K		K		K		K			K		K	
Tier 2 NRN							S			S			
Tier 2 NNF					K								
Tier 2 NNN			S		K+S		S			S		S	
Tier 2 TNF					K								
Tier 2 TNN	K		K		K		K			K		K	
Tier 2 BRF					K		V			V			
Tier 3 NRF	K+S		K+S		K+S		K+S			K+S		K+S	
Tier 3 TRF	K+S		K+S		K+S		K+S			K+S		K+S	
Tier 3 TRN	K		K		K		K			K		K	
Tier 3 BRF					K		V			V			
Tier 3 TNF	K		K		K		K			K		K	

C | Schema Evolution Analysis

I selected a set of structural and type conversion schema transformations and tested if they result in compatible changes using the schema-driven serialization implementations and encodings introduced in Table 5.1. A different encoding of the same schema-driven serialization specification may yield different results. The results of the structural schema transformations are presented in Table C.2 and the results of the type conversion schema transformations are presented in Table C.3. I mark the test results as shown in Table C.1.

I found that sometimes the schema evolution features of a serialization specification are subtly affected by the data types being used and by the infinite possibilities of surrounding data. For this reason, I recommend schema-writers to use these results as a guide and to unit test the schema transformations they plan to apply before deploying them. I also encountered various cases of undocumented compatible schema transformations. These transformations may rely on accidental behaviour of either the serialization specification design or the chosen implementation and may carry no future guarantees. I encourage readers to consult the official schema evolution documentation and check if their serialization specification of choice satisfies the intended compatible transformation by design or by accident.

Table C.1: Descriptions of how I will mark schema evolution transformation results.

Symbol	Description	When
A	Fully-compatible	The schemas are fully-compatible for all tested instances
F	Forwards-compatible	The schemas are forwards-compatible for all tested instances, despite backwards-compatibility failures or exceptions
B	Backwards-compatible	The schemas are backwards-compatible for all tested instances, despite forwards-compatibility failures or exceptions
N	Silently-incompatible	The schemas are not forwards nor backwards-compatible in at least one tested instance but no exception is thrown
X	Runtime exception	The schemas are neither forwards nor backwards-compatible for all tested instances and at least one exception is thrown
	Not-applicable	The schema transformation is not applicable to the serialization specification as it involves data types not supported by the serialization specification

Description from Table C.2.

(1) Microsoft Bond [96] supports the concept of *required_optional* fields that are required at serialization time but optional at deserialization time. This concept enables schema-writers to make an optional field required and viceversa in a fully-compatible manner through a two-step process: Changing an optional or required field to *required_optional*, deploying the schema update to both producers and consumers, and then changing the *required_optional* field to required or optional.

(2) ASN.1 PER Unaligned [124] and Cap'n Proto [146] support updating a list of scalars to a list of structures where the scalar is the first and only element in a fully-compatible manner. In the case of ASN.1 PER Unaligned, this transformation is possible because structures are list of values and a list of structures with a single required scalar is encoded in the same manner as a list of such scalars. In the case of Cap'n Proto, a list definition declares whether its element are scalars or composites as shown in [151]. If the elements are composite, the list definition points to a 64-bit word that defines the composite elements, allowing the deserializer to determine if following the pointer or not yields a scalar of the same expected type. As an exception, Cap'n Proto does not support this schema transformation on a list of booleans for runtime-efficiency reasons [1].

(3) Protocol Buffers Binary Wire Format [67] supports transforming a field into a list of a compatible type in a backwards-compatible manner. Protocol Buffers Binary Wire Format encodes lists as multiple occurrences of the same field identifier or as a concatenation of the members prefixed with a length-delimited type definition in the case of packed field encoding. This design decision makes implementations using the new schema interpret a standalone value as a list consisting of one value.

[1] https://capnproto.org/language.html#evolving-your-protocol

(4) Serialization specifications based on field identifiers that implement unions without involving additional structures such as Protocol Buffers Binary Wire Format [67] support forwards-compatibility when moving an optional field into an existing union. In this case, union choices and fields outside of the union share the same field identifier context. This means that an application using the older schema either leaves the union choices or the optional field outside the union unset. In comparison, FlatBuffers [143] requires creating a new data structure to hold the union type. As a result, it supports backwards-compatibility when moving an optional field into an existing union as an application using the newer schema will ignore the optional field outside of the union. The converse is true when extracting an optional field out of an existing union.

(5) Protocol Buffers [67] implements unions based on field identifiers on the current identifier context and supports unions of a single choice. Therefore, an optional field and a union of the single field are equivalent. A similar argument follows for Cap'n Proto [146], however Cap'n Proto does not support unions of a single choice, making this transformation only backwards-compatible. Apache Avro [61] supports unions of a single choice, however its schema resolution rules throw an exception on the forwards-compatible case.

Table C.2: A schema transformation result is annotated as shown in Table C.1. The *Type* column documents whether a schema transformation confines (`C`), expands (`E`), changes (`!`), or preserves (`=`) the domain of the schema.

Category	Type	Schema Transformation	ASN.1 PER Unaligned	Apache Avro Binary Encoding	Microsoft Bond Compact Binary v1	Cap'n Proto Packed Encoding	FlatBuffers Binary Wire Format	Protocol Buffers Binary Wire Format	Apache Thrift Compact Protocol
Structures / Tables	E	Add an optional field to the end	A	A	A	A	A	A	A
	C	Remove an optional field from the end	A	A	A	A	A	A	A
	C	Add a required field	F	F	F		F		F
	E	Remove a required field	B	B	B		B		B
	C	Optional to required	F	F	A (1)		F		F
	E	Required to optional	B	B	A (1)		B		B
	!	Change field default	N	N	N	N	N		N
Lists	E	List of scalars to list of structures with scalar	A (2)	X	N	A (2)	X	X	X
	E	Scalar to list of scalars	X	X	X	N	N	B (3)	N
	E	Composite to list of composites	X	X	X	X	X	B (3)	N
Unions	!	Move optional field to existing union	X	X		N	B (4)	F (4)	N
	!	Extract optional field from existing union	X	X		N	F (4)	B (4)	N
	E	Move required field to existing union	X	X			B		N
	C	Extract required field from existing union	X	X			F		N
	!	Move optional field to a new union	X	B (5)		B (5)	N	A (5)	N
	E	Move required field to a new union	B	B			N		N
	E	Add choice to existing union	B	B		B	B	B	B
	C	Remove choice from existing union	F	F		F	F	F	F
Enums	C	Scalar to enumeration	F	X	F	F	F	F	F
	E	Enumeration to scalar	B	X	B	B	B	B	B
	E	Add enumeration constant	B	B	B	B	B	B	B
	C	Remove enumeration constant	F	F	F	F	F	F	F

Table C.3: A schema transformation result is annotated as shown in Table C.1. The *Type* column documents whether a schema transformation confines (`C`), expands (`E`), changes (`!`), or preserves (`=`) the domain of the schema.

Category	Type	Schema Transformation	ASN.1 PER Unaligned	Apache Avro Binary Encoding	Microsoft Bond Compact Binary v1	Cap'n Proto Packed Encoding	FlatBuffers Binary Wire Format	Protocol Buffers Binary Wire Format	Apache Thrift Compact Protocol
Type Conversions	E	Increase integer width	N	B	B	N	B	B	N
	C	Decrease integer width	N	F	F	N	F	F	N
	E	Increase float precision	B	B	B	N	N	N	
	C	Decrease float precision	F	F	F	N	N	N	
	E	Unsigned to larger signed integer	N		X	N	B	B	
	C	Signed to smaller unsigned integer	N		X	N	F	F	
	E	Signed integer to float	X	B	X	N	N	N	N
	C	Float to signed integer	X	F	X	N	N	N	N
	E	String to byte-array	B	X	X	B	B	B	B
	C	Byte-array to string	F	X	X	F	F	F	F
	E	Boolean to integer	X	X	X	N	B	B	N
	C	Integer to boolean	X	X	X	N	F	F	N
	=	Byte-array to array of 8-bit unsigned integers	A		A	A			

D | Feedback

Initial feedback received from the community after publishing the papers *A Survey of JSON-compatible Binary Serialization Specifications* [151] and *A Benchmark of JSON-compatible Binary Serialization Specifications* [150] is shown in Figure D.1, Figure D.2, Figure D.3, Figure D.4 and Figure D.5.

Figure D.1: Wouter van Oortmerssen, the primary author of FlatBuffers [143] and FlexBuffers [144] at Google, has shared the paper *A Survey of JSON-compatible Binary Serialization Specifications* [151] on Twitter [1].

Figure D.2: An excerpt from an e-mail sent by Wouter van Oortmerssen, the primary author of FlatBuffers [143] and FlexBuffers [144] at Google after the publication of *A Survey of JSON-compatible Binary Serialization Specifications* [151].

Figure D.3: Feedback from a member of the MessagePack [64] community on GitHub [2].

[1] https://twitter.com/wvo/status/1480636224339853314?s=20
[2] https://github.com/msgpack/msgpack/issues/316#issuecomment-1012651688
[3] https://lists.apache.org/thread/r6o9lx4jhowob0xrjlc8z5wym7vltjy3
[4] https://lists.apache.org/thread/nfl981yhnvs756lqyknjx8f26vdpfnkn

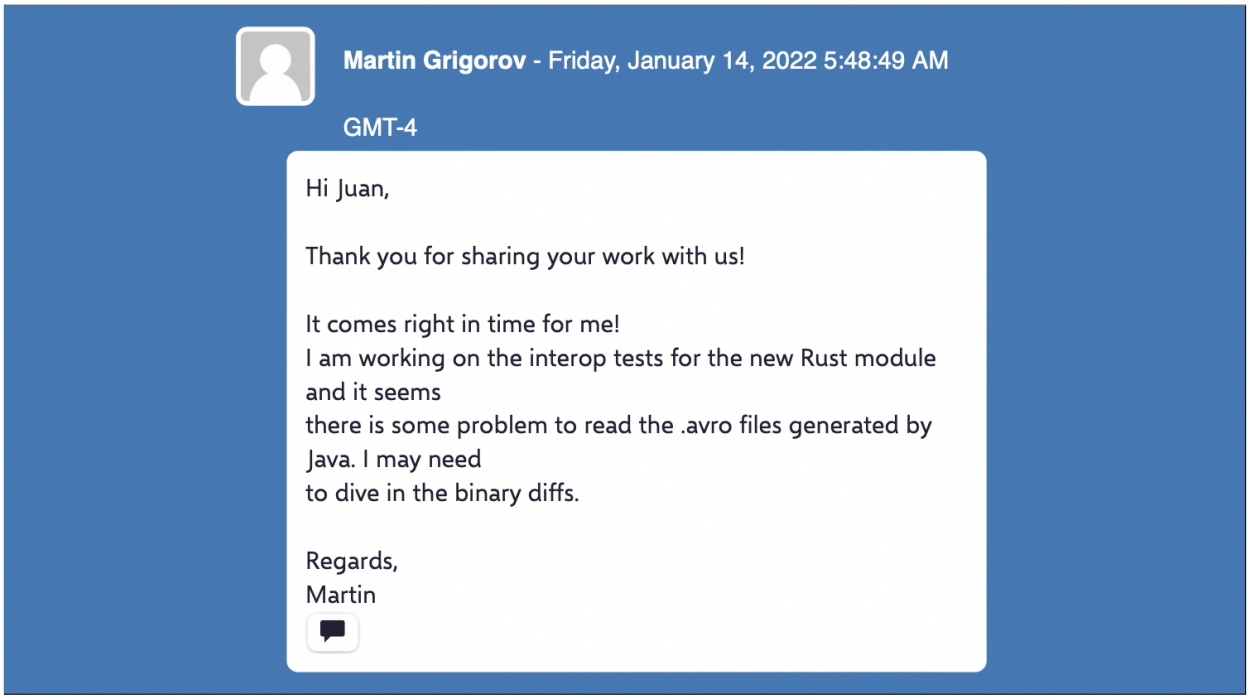

Figure D.4: Feedback from a member of the Apache Avro [61] community on the users mailing list [3].

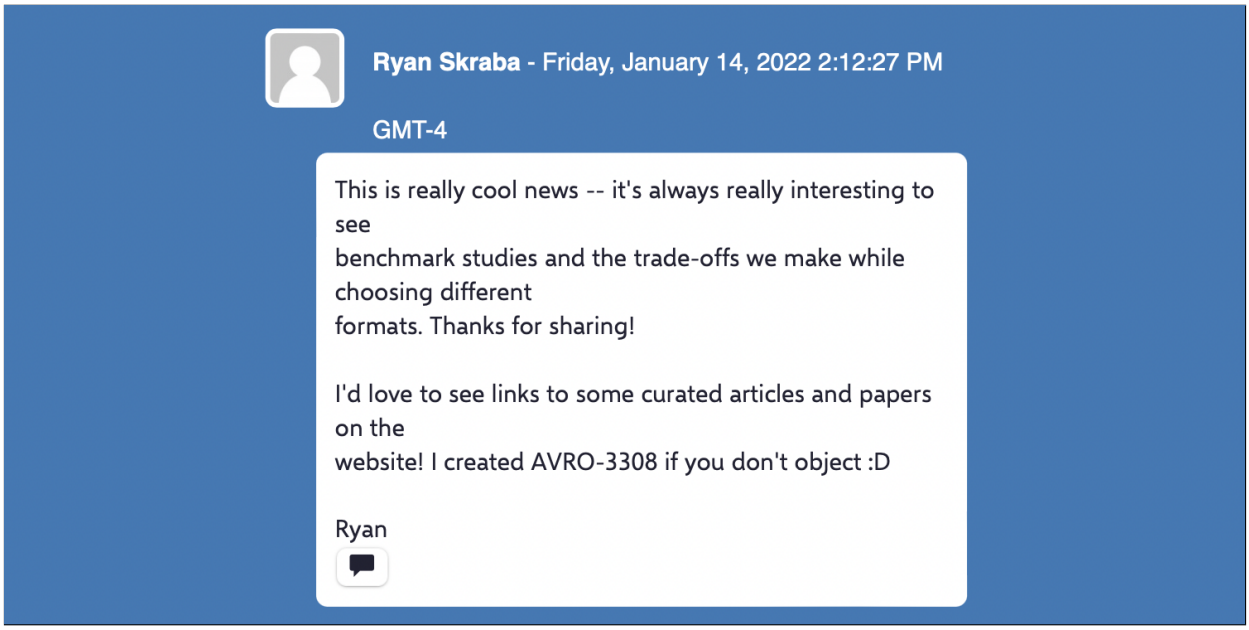

Figure D.5: Feedback from a member of the Apache Avro [61] community on the users mailing list [4].