An Analysis of Defects in Public JSON Schemas

Claire Yannou-Medrala and Fabien Coelho
firstname.lastname@minesparis.psl.eu
Centre de recherche en informatique, Mines Paris – PSL University
France

ABSTRACT

JSON is a simple *de facto* standard cross-language textual format used to represent, exchange and store data and documents in computer systems. JSON Schema is a description language, based on JSON, proposed to describe JSON types and validate JSON data. We investigate over 57,800 distinct public schemas for various defects through static analysis, and identify cases of mistyping, misplacing, misnaming, misspelling, misversioning and other miscellaneous issues. Over 60% of schemas are defective, allowing in the worst case unintended data to be validated. These findings suggest to make key changes to the current JSON Schema draft so as to limit potential issues. It also leads us to design JSON Model, an alternative compact and expressive JSON data structure description language.

KEYWORDS

JSON Schema; Static Analysis; JSON Model.

1 INTRODUCTION

The JSON [16] *JavaScript Object Notation* format has become in recent years an ubiquitous cross-language *de facto* standard to represent, exchange and store data between computer applications, partially replacing XML [23]. Its success can be attributed to the extensive use of JavaScript in web and mobile development. Like the more verbose XML, JSON can be parsed without knowing in advance the expected structure. It allows to serialize in textual form simple data structures (Figure 1) built upon the *null* value, booleans, numbers, Unicode strings, arrays (aka list, tuple, sequence, set) and objects (aka struct, record, dict, map, association, key-value pairs). Its drawbacks include the limited number of types, the absence of a syntax for comments, its unbounded numbers which cannot express some values (e.g. *NaN*), the restriction of object properties (aka key, attribute, field) to strings, and that only tree structures can be serialized, i.e. there is no sharing of values or cycle. Thanks to these simple features, a wide range of libraries and tools are available for many programming languages and systems beyond JavaScript [18] including Python, Java, Shell, and SQL.

```json
{
  "name": "Calvin",
  "age": 6,
  "friends": [ "Hobbes", "Susie" ]
}
```

Figure 1: JSON object with three properties

Several systems have been proposed to declare JSON types so as to provide a mean for documenting expected structures and validating JSON data beyond their mere syntax. Some systems are based on JavaScript [8, 27], while others rely on JSON itself [1, 29]. We focus here on JSON Schema [29], a JSON syntax (see Figures 2 and 3) which describes JSON types. It has been developed as a potential standard for over 13 years and is still under discussion.

In this paper, we investigate the quality of schemas. We analyze over 57,800 distinct public schemas through static analysis complemented with manual checks, looking for typical defects left by schema designers. Defective schemas do not reflect the intended data structure, possibly allowing invalid data into a system, and may have disastrous consequences such as cybersecurity issues or unexpected system failures. For each defect, we try to guess the quite often obvious designer initial intention. We argue that defects are mostly due to key design decisions in the current standard draft, which should be revised in the light of these findings.

Our contributions include: (1) an in-depth study of JSON Schema specifications for potential issues that can lead to defects; (2) the collection of over 57,800 distinct public schemas for our investigations made available online [30]; (3) the development of proof-of-concept tools for processing JSON Schema and detecting various categories of common defects, also available online [12, 14]; (4) a categorization of the causes of defects found by our tools on the corpus; (5) based on these evidences, a set of recommendations for improving the next version of the JSON Schema specification; (6) the outline of JSON Model, an alternative compact and expressive JSON-based data description language for JSON.

The remainder of this paper is organized as follows: Section 2 surveys prior empirical studies we built upon, as well as some theoretical contributions about JSON Schema. Section 3 then presents a comprehensive summary of the current state of the JSON Schema specification, outlining specific features likely to lead to defects. Section 4 describes the various sources, the preprocessing stages and the tools used for our schema collection and analysis. Section 5 outlines the broad set of defects found in our corpus. This leads us to recommend changes for improving the JSON Schema specification in Section 6, and to propose, in Section 7, JSON Model, an alternative compact and expressive data description language, before concluding in Section 8.

2 RELATED WORK

Prior studies have focused on analyzing the schemas of databases or documents from public corpora, either to gather insights about actual schema usages or to look for defects. Relational database schemas from open-source projects have been analyzed [11] to look for common design issues such as missing primary keys, or how often foreign keys are (not) declared. Non-relational (NoSQL) databases schemas extracted from Java applications have been studied [26] to check for denormalization and measure the frequency of schema changes. The schemas of XML documents have also been a subject of study, e.g. a sample of DTDs has been investigated [10] for their complexity, determinism and ambiguities, as well as properties such as reachability, recursion or cycles.

Recent studies have looked in-depth at available JSON Schema samples: 159 schemas from Schema Store, a curated corpus of schemas, have been inspected [22] to collect evidence about their sizes, distribution of types, looseness, the usage of recursion and the maximum level of nesting depths. Over 8,000 pairs of schemas have been investigated [20] to check for subschema relations between subsequent versions: the *irrelevant keyword* rewriting rule of the canonicalization phase often corresponds to a misplaced keyword (Section 5.2). The evolution of 230 schemas from the SchemaStore has been analyzed [19] to check for schema change compatibility with the use of comparison tools. Finally, over 80,000 schemas from GitHub have been collected and looked into [6] for keyword usages, especially patterns of negation.

JSON Schema has also received theoretical attention: a semantics has been proposed [25]; a formal data model and a query language have been defined [9]; the validation of schemas has been formalized and its complexity asserted [3]. Other works focus on interactive type inference [4], automatic data generation from schemas [2, 15] or the reverse, algorithms to generate a schema from raw JSON data [28]. Theoretical works point to the complexity of JSON Schema when handling schemas and validating instances: PCRE regular expressions with backtracking implementations can result in exponential execution time when looking for a match [24]; because of dynamic references and recursion, the verification complexity of JSON Schema is PSPACE-complete [3]; negation has been investigated [7] in relation with other operators and shown not to be closed, i.e. the **`not`** keyword cannot always be removed.

3 JSON SCHEMA

The JSON Schema specification [29] was first proposed as an IETF Internet Draft in 2009. Since then, 10 versions have been published, with significant changes on the way, such as renaming keywords or modifying their purpose and type, but also removing or even resuming features, so that there is no compatibility between versions. This lengthy and chaotic development history induces confusion for tool developers and early adopters alike.

3.1 Design Philosophy

Three key design choices permeate the JSON Schema draft standard: (1) JSON is the foundation; (2) JSON is used to represent loose *documents*; (3) schemas are *themselves* loose.

The first choice means that JSON Schema describes JSON with JSON. This is useful for such a language to gain acceptance as a cross-language tool, and also simplifies tool development. Because of this choice, a schema has a *meta schema* written as a schema, which can be validated against itself.

The second choice is to assume that JSON is used to represent large evolving *open documents*, somehow like XML, as opposed to small rigid *data* which are used inside or at the interface of a programming language to exchange data. A concept underlying this point is that schema definitions can be either *tight* or *loose*: a tight schema does not allow much freedom about properties or types, whereas a loose schema allows unknown properties. As JSON schema aims at documenting in detail the structure and the semantics of the various data as much as constraining their types, it incurs a level of verbosity and scafolding in the language. Moreover, because these documents are assumed to be evolving, all defaults are chosen so that unknown elements in a document are accepted, i.e. schemas are loose by default.

The third choice is that JSON Schema *meta schema* is loose. This is a pernicious effect of the above open document approach applied to the specification of JSON Schema itself. Choosing a loose meta schema allows extensions without updating existing tools, but also induces that any keyword misspelling or misplacement is coldly ignored. As the point of schemas is to catch errors in data, it is paradoxical that they would be error-prone to write.

3.2 Schema Structure

With the exception of booleans used to represent *any* or *no* possible values, all elements of a JSON data structure are represented in a schema as independent nested objects, as shown in Figure 2. In this example, the outside `Person` structure is an object which contains three **`properties`**. The properties `name` and `age` are simple types with constraints, expressed as sub-schemas. The property `friends` is an array of strings, inducing two additional levels of sub-schemas. Only the first two properties are mandatory (**`required`**), and any other property is rejected (**`additionalProperties`**). Overall, this schema description is over 4 times larger than a sample value.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "A Person",
  "type": "object",
  "properties": {
    "name": { "type": "string", "minLength": 1 },
    "age": { "type": "integer", "minimum": 0 },
    "friends": {
      "title": "The Person's Friends",
      "type": "array",
      "items": { "type": "string", "minLength": 1 },
      "minItems": 1
  } },
  "required": [ "name", "age" ],
  "additionalProperties": false
}
```

Figure 2: Tight JSON schema for Figure 1

JSON Schema latest specification, version 2020-12, defines 60 camel case keywords, their expected types and associated semantics. They cover meta data, type constraints, assertions about values or sizes, subschema logic and consistency constraints. From a software engineering and practical point of view, a schema description language with 60 keywords, including logical structures, seems rather cumbersome. It is especially so when considering that JSON itself has only 3 keywords (**`null true false`**) and 8 syntactic markers (`{} [] \ " , :`) or compared to full-fledged programming languages such as C (32 keywords), Python (34), JavaScript (46), TypeScript (47) or Java (51).

Structural nesting is expressed as elements which are schemas themselves when appearing within some keywords. All keywords can appear at any level of the schema, without any consistency constraint through *keyword independence*: the validation semantics is lax, as keywords which do not apply to the current value are

silently ignored. Some keywords allow to combine an array of schemas logically: **allOf** requires that the current value matches all subschemas, **anyOf** checks that it matches at least one of them, and **oneOf** validates that only one matches.

When validating a JSON value (a process called *evaluation of an instance against a schema* in JSON Schema specification), the system considers keywords directly inside a JSON Schema element (adjacent keywords), but also those specified in subschemas through combinators (**allOf anyOf oneOf**) and references (**$ref**), so that in effect these subschemas apply to the current value as well. Moreover, the system must dynamically keep track of values (e.g. properties) not validated by subschemas to possibly apply special catch-up checks specified through keywords **unevaluatedItems** and **unevaluatedProperties**. This evaluation semantics is complex to understand because it involves several schema levels at the same time, as illustrated later in Figure 22.

3.3 Permissiveness

Constraints declared in a schema are not necessarily enforced depending on the context, because of permissive default values.

A first permissive setting is that the **type** keyword is not mandatory, and its absence means that no check is performed about the value type, conducting to accept any value. A second permissive setting is that when properties are not listed they are allowed by default, otherwise they must be *explicitly* forbidden with **additionalProperties**. The same holds for array items which are of any type by default. A third permissive setting is that properties are optional by default, and they must be explicitly **required**.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "A Person",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "integer" },
    "friends": {
      "items": { "type": "string" }
} } }
```

Figure 3: Loose JSON schema for Figure 1

Figure 3 provides a loose schema for our JSON data introductory example in Figure 1. It is much simpler as it involves less concepts (7 vs 16 keywords): the resulting size is about half of the tight version. This loose schema validates all data in Figure 4. In Extract (a) Object Hobbes: the `nom` (French for `name`, possibly misspelled), `species`, `color` and `birth` properties are simply accepted as the document is open; the `friends` property is not an array so the **items** constraint is ignored; the only actual verification is that `age` is some sort of integer. Worst, because of missing **type** declarations, it also validates all seemingly unrelated miscellaneous data in Extract (b): what the schema actually validates is that *if* the data is an object *and* it has `name`, `age` or `friends` array properties, *then* they must be a string, an integer and composed of strings, respectively.

The attentive reader will have noticed that, thanks to JSON Schema lax validation semantics, Calvin and Hobbes JSON objects in Figures 1 and 4 (a) are both *valid schemas*, as none of the 7 property names collides with any of the 60 keywords defined by the

```
{
  "nom": "Hobbes",
  "age": 3.0,
  "friends": "Calvin",
  "species": "tiger",
  "color": "yellow ochre",
  "birth": "1985-11-18"
}
```

(a) Object Hobbes

```
{
  "comics": "Calvin & Hobbes",
  "creator": "Bill Watterson"
}

"Calvin and Hobbes are Friends"

3.14159265358979323846264338327
```

(b) 3 Miscellaneous Valid Values

Figure 4: Valid JSON instances with Figure 3 schema

specification. Because unknown properties are ignored, when considered as schemas they are equivalent to `{}` which means that all possible JSON values are valid. One step further, as schemas, they have the notable property of being self-validating. The practical implication of these twisted remarks is that if a user loads a random JSON object file as a schema it is likely to be accepted and indeed validate all its data, so could go unnoticed.

3.4 Version Changes

The lengthy and yet uncompleted development of the JSON Schema specification is quite chaotic, with significant incompatibilities between versions. Although it is not a requirement for successive *drafts* to be compatible, the development of well over 150 tools [21] shows a real demand for a standard, and these continuous changes hinder tool maintenance: a fraction of them have not followed the latest versions published over 3 years ago.

A very minor but still emblematic nitpick is that draft version identification has changed from straightforward numbers (drafts 0 to 8) to year-month, with version 8 also identified as 2019-09. This later system hints at more years of specification to come. The latest planed version has been dubbed 2022-XX, but was not released before 2023, following the trend started with Fortran 8X which ended as Fortran 90. A less minor change is the definition of *integer*, which was relaxed from *number without a decimal part* (up to v5) to *number with a zero decimal part* (from v6), so that `2.0` became an integer overnight.

Keyword types and behaviors have changed through the drafts:

- Types were initially recursively defined, then later restricted to a predefined list of string values, which lost the redundant **any** special type at some later point.
- Initial versions had mandatory properties which could be changed to **optional** with a boolean, then properties became optional and could be made **required** with a boolean, then **required** became a list of strings outside of the property definitions. This array could not be empty up to v5, then this constraint was relaxed.
- Boolean **minimumCanEqual** was inverted to **exclusiveMinimum** which was switched later to a number.
- **contentEncoding** disappeared in v3 to reappear in v7.
- Definitions were implicit in earlier drafts thanks to transparent nesting, then were made explicit with **definitions** in v4, which was then changed to **$defs** in 2019-09.
- The **id** keyword was renamed **$id** in v6.

- The management of arrays was turned upside down between 2019-09 and 2020-12, with **`items`** and **`additionalItems`** becoming **`prefixItems`** and **`items`** respectively, changing as a side effect the semantics of **`items`** and breaking the style consistency with **`additionalProperties`**.
- New keywords can last just one version: **`recursiveAnchor`** introduced in 2019-09 was renamed **`dynamicAnchor`** in the next version.

The predefined **`format`** values for strings have seen a number of apparently random changes, such as `utc-millisec` which is only defined in v3; `host-name` was renamed to `hostname` in v4; the trivial `date` and `time` disappeared in v4 to reappear in v7; this is also the case for `regex`, even if it is mandatory for defining a neat meta schema; version 5 is advertised as a simple rewrite of v4, but really introduced `uri-ref` which is then renamed `uri-reference` in v6. Most notably, formats which would be definitely useful in any programming language beyond JavaScript are absent: there are no `int32` `int64` `float` `double` or similar predefined formats. A source of confusion is that two keywords convey a type information, with a subtle distinction: **`type`** is about the type from a JSON perspective, whereas **`format`** is about type from the application perspective.

Minor naming inconsistencies can also be found: expressing a minimum in various contexts is shown explicitly (**`minContains`** **`minItems`** **`minLength`** **`minProperties`**), but the minimum for a value is **`minimum`**, whereas the homogeneous **`minValue`** would have made sense.

Section 5.4 shows various defects which demonstrate that schema designers have had a hard time to keep up with these confusing changes.

4 DATA COLLECTION

We have collected JSON Schema samples by reusing corpora from previous studies [6, 19, 20, 22] and adding our own directly collected samples to the mix. After some preprocessing to remove duplicate schemas, we have run a set of automatic analysis tools to extract relevant information.

4.1 Schema Collection

Our corpus [30] is available online. It comes from 5 main sources:

Ref covers JSON Schema Test Suite (over 2000 extracted files), some examples schemas and JSON Schema own meta schemas;

Store includes the Schema Store, a curated corpus of schemas, and Schema-Store-Analysis, a dated extract used in [22];

ODS are schemas collected from the self-service OpenDataSoft *explore* interface of Open Data platforms, including nearly 30,000 from ODS data hub; these schemas are dynamically generated from an internal JSON representation;

JSC The json-schema-corpus [5] comprises 80,000 files collected from GitHub repositories in July 2020; for some files which were not actual schemas but happened to contain schemas, we extracted the subschemas, adding over 1600 samples;

Misc includes various sources, the largest of which is the Kubernetes collection (over 400,000 files, of which we kept the *standalone-strict* variant among the four available variants), schemas from French public open data sites, and others;

These schemas cover very diverse application domains, such as mobility, geography, education, IT, media…we aimed at collecting all possible public schemas without any filtering, thus we believe that our collection is representative of public JSON schemas.

Source	Name	Files	Schemas	%
Ref	JSON Schema Test Suite	2,149	1,990	92.6
	JSON Schema Specification	37	35	94.6
Store	Schema Store	516	512	99.2
	Schema Store Analysis	165	141	85.5
ODS	Data Hub	27,816	21,808	78.4
	Others…	2,488	1,372	55.1
JSC	JSON Schema Corpus	82,094	22,427	27.3
	extracts	11,994	1,661	13.8
Misc	Kubernetes	102,582	6,145	6.0
	Washington Post	3,239	634	19.6
	Others…	2,297	1,078	46.9
		235,377	57,803	24.6

Table 1: Corpus Sources, retrieved in September 2023

Table 1 shows detailed counts about our corpus. Overall, our analysis operates on around 235,000 raw valid JSON files, including many duplicates, so that only about 57,800 distinct schemas are kept.

4.2 Preprocessing

We first preprocess all files to remove duplicates identified by hashing a normalized JSON representation. The normalization consists in simplifying titles, definitions and references from the ODS source, and pretty-printing the JSON data with sorted object properties. Finally, a further cleaning phase removes over 1000 more files from the JSC source which, despite vaguely looking like schemas, e.g. they have an official **`$schema`** URL as shown in Figure 5, have nothing to do with JSON Schema. This preprocessing keeps one fourth of the initial corpus. These files are further analyzed automatically.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "asd",
  "name": "Audio Side Data",
  "defaults": { },
  "groups": [ ]
}
```

Figure 5: Rejected Schema – JSC PP 16421

4.3 Static Analysis

We have developed a set of Python tools [12, 14] to analyze schemas from any version and report basic statistics, validation results against all meta schema versions (described with JSON Model, Section 7), as well as detected defects. These analyses take on average 7 ms per schema. The result is imported into a Postgres database for further processing with SQL queries.

Figure 6 shows schema numbers per size from our corpus. There is a very wide spread of schemas sizes, from 4 bytes (**`true`**) to about

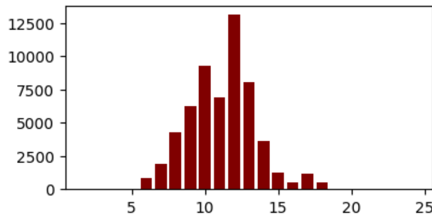

Figure 6: Number of schemas per log2 JSON sizes

15 MB (2^{24} bytes). The median size is 2.5 kB (2^{11}) but the average is about 11 kB, because of a few very large files in our corpus.

We collect simple structural JSON statistics (number of nulls, booleans, strings, arrays, objects, properties…). When considered as JSON data, the type of the elements found in schema description is largely skewed towards strings (74.7%) and objects (17.3%), because all schema elements are described with keyword properties inside objects. Arrays are infrequent (5.7%), booleans are rare (1.6%). Few numerical values are used in schemas (0.6%), mostly integers for various seldom-used constraints (string length, array sizes…).

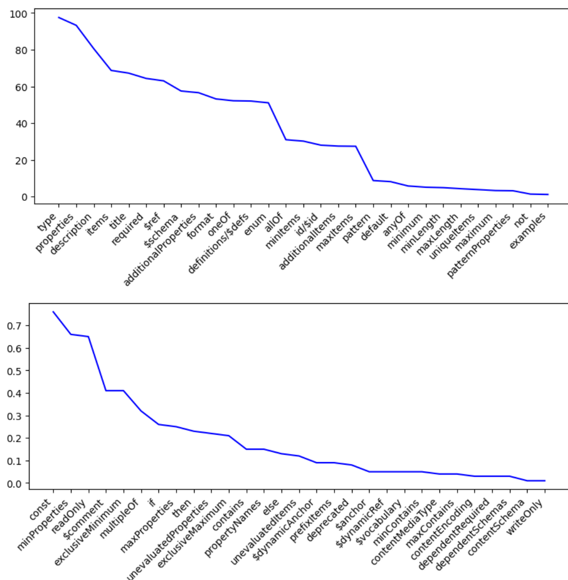

Figure 7: Percent of schemas using keywords

Then we compute JSON schema specific statistics (e.g. which keywords are used and how often they are used), and for some keywords statistics about the values (e.g. which schema version is advertised if any). Figure 7 shows the percentage of schemas using each keyword: the upper plot includes keywords which appear in over 1% of schemas, the lower plot those under 1%. To smooth version differences, occurrences of **`id`** and **`$id`** have been merged, as well as **`definitions`** and **`$defs`**. Consistently with [6], **`type`** is the most used keyword, however there is still 2.5% of schemas without it, mostly coming from the official test suite. The second keyword is **`properties`**, reflecting that schemas usually contain at least one object. 80% of schemas use **`description`**, which underlines the documentary nature of a lot of schemas in our corpus.

Definitions (**`definitions $defs`**) appear in 52% of schemas, however they are not always used: over half of schemas with definitions have at least one unused definition, mostly from the ODS source. When looking at schema keyword usage, some are seldom used, mostly in tests, and do not seem to reflect significant use cases: they include **`minProperties`**, **`multipleOf`**, **`maxProperties`**, **`contains`**, **`contentMediaType`**, **`contentEncoding`**.

Next, we validate the schema against all meta schema versions through our JSON model compiler [13], with separate tests for *tight* vs *loose* versions of the meta schemas, and for early versions whether schema nesting is allowed. We also check whether keyword usage in the schema is compatible with its advertised version.

Finally, our tools perform static analysis to look for specific defects and common typos. The main analyses, which detect most defects, are:

missing type an element has no associated type, despite some keywords suggesting a particular type.
dangling reference a reference does not have a definition.
dead keyword a keyword has no effect on the current value because of possible types at this point.
draft incompatibility and miscellaneous version issue.
type error such as invalid or empty type.

For each defect found, the JSON file and the path leading to the defective element is shown, along with details about the issue. These findings are presented in the next section.

5 SCHEMA DEFECTS

Most defects manifest themselves as inconsistencies coming from:

(1) an untyped element with keywords which suggest that it is expected to be typed (missing type, as in Figure 3); this is the most common error which impacts 53% of our corpus;
(2) a known keyword without any effect on the expected values because the possible types at this point, for which there may be several explanations which must be investigated by hand: the expected type may be wrong and the keyword is right (typing issue); the type is right but another keyword was meant (misnaming); the keyword is right but should be at another level (misplacement) and possibly needing another intermediate keyword (missing);
(3) a known keyword ignored because of its placement, independently of the type, typically a keyword used in property declarations;
(4) an unknown keyword (at least for the advertised version) or unexpected value leading to an inconsistency, which is quite often a typo (misspelling) or possibly an nonexistent feature, at this version (misversioning or mistake).

Note that static analysis will find defects because of the implications on the semantics: if a schema does not reflect the intention of the designer but is valid, our analyses cannot find it. Also, the PoC nature of our tools means that we can have some false positives. In the remainder of this section, we describe the different defects and illustrate them with simplified examples actually extracted from our corpus, mostly from the curated Store source. The source file and path of all reported defect extracts are available in [30].

5.1 Mistyping

Ultimately, defects are linked to type errors. In this section, we outline typical cases where `type` declarations either implicit (`enum` `const` `default` `examples`) or explicit (`type`), direct (in the object) or indirect (through references or combinators), are probably wrong or just totally missing. They explain why some keywords are without effect on the validated values.

```
{
  "type": "string",
  "enum": [ 80, 443 ]
}
```

(a) Store – Ansible 2.5

```
{
  "type": "string",
  "enum": [ [], {} ]
}
```

(b) Store – Style Lint

```
{
  "type": "object",
  "oneOf": [
    { "type": "string" },
    { "type": "number" }
  ]
}
```

(c) Misc – K8S Quantity

```
{
  "type": "array",
  "anyOf": [
    { "type": "object" },
    { "type": "object" }
  ]
}
```

(d) Store – Bamboo

```
{
  "type": "object",
  "enum": [ "azure.cloudify_azure.resources.network..." ]
}
```

(e) Store – Azure Cloud

Figure 8: Type Errors

When explicitly provided, type declarations can be given a list of types, which makes schemas confusing as assertions about incompatible types can legitimately be intermixed inside one element. It is also possible to build a JSON element which cannot accept any value because of contradictory type requirements, as shown in Figure 8: the types suggested by `enum`, `oneOf` and `anyOf` are incompatible with the explicit `type` declarations. This later declarations should probably be removed from all 5 extracts so that the expected values are validated.

```
{
  "type": "object",
  "items": {
    "type": "object",
    "properties": {
      "ids": { }
    }
} }
```

(a) Store – Google Chrome

```
{
  "title": "Default",
  "type": [
    "string", "number",
    "integer", "boolean", "array"
  ],
  "additionalProperties": false
}
```

(b) JSC – PP 27224

```
{
  "type": "integer",
  "pattern": "/[1,-1]{1}/"
}
```

(c) JSC – PP 24218

```
{
  "type": "number",
  "pattern": "[1-9][0-9]*"
}
```

(d) JSC – PP 2512

Figure 9: Other Type Errors

```
{
  "type": "object",
  "default": null
}
```

(a) Store – Cypress

```
{
  "type": "integer",
  "examples": ["27017"]
}
```

(b) Store – Monika

Figure 10: Type inconsistency

In Figure 9, some keywords can only be useful if the type is changed: Extract (a) `items` suggests that the element should actually be an array. Extract (b) `additionalProperties` hints that the element could be an object. Extracts (c, d) `pattern` is ignored on numbers, so either a string was really expected (mistyping), or the pattern must be removed (mistake). Finally, Figure 10 shows inconsistent `default` (`null` should be quoted) and `examples` (the integer should be unquoted).

```
{
  "required": [ "vsixId" ],
  "properties": {
    "name": { },
    "vsixId": { }
  }
}
```

(a) Store – Vs Ext

```
{
  "type": "object",
  "properties": {
    "instructions": {
      "description": "An array..."
    }
} }
```

(b) Misc – LaunchDarkly

Figure 11: Missing Types

Another common typing error is that type declarations are missing, so that even if keywords suggest that a particular type is expected, any other unconstrained value of other types can be inserted. This is illustrated in Figure 11: In Extract (a) the `required` and `properties` keywords suggest that an object is really expected, but without an explicit `type` all other types are allowed. In Extract (b) there is no type hint as such, but the `description` says that an array is expected. Such later cases are hard to distinguish from rare but legitimate untyped elements.

In later Figure 14 (e), beyond the misspelling of extrema, the number type which denotes an HTTP status code should really be an integer as hinted by the erroneous `integer` keyword.

A number of apparent type errors are really misspellings or mistakes, or due to recurrent `type`/`format` confusions, as discussed in Section 5.3.

5.2 Misplaced Properties

Because of *keyword independence*, properties which do not apply to the current value are ignored. Data valid with respect to the schema designer intentions will still be valid anyway. As the result, it is very easy to put and let indefinitely a property at a wrong place.

In Figure 12 Extracts (a, b), the `uniqueItems` array property is associated to the array items instead of the array itself. It should be moved one level up. Extract (c) `propertyNames` is put on the additional property string, instead of the enclosing object, thus is ignored. Extracts (d-f) `properties` contain suspicious keywords as user-defined properties: `anyOf`, `additionalProperties` and `minProperties` probably belong to the containing object.

```
{
  "type": "array",
  "items": {
    "type": "string",
    "uniqueItems": true
  }
}
```

(a) Store – .NET Template

```
{
  "type": "array",
  "items": {
    "type": "object",
    "uniqueItems": true,
    "properties": { }
} }
```

(b) Store – UI5 Manifest

```
{
  "type": "object",
  "properties": { },
  "additionalProperties": {
    "type": "string",
    "propertyNames": {
      "maxLength": 10
} } }
```

(c) Store – Azure Device

```
{
  "type": "object",
  "properties": {
    "anyOf": [
      { "$ref": ". . . " },
      { "$ref": ". . . " }
    ]
} }
```

(d) JSC – PP 81088

```
{
  "type": "object",
  "properties": {
    "image": { "type": "string" },
    "additionalProperties": false
} }
```

(e) Store – Fly

```
{
  "type": "object",
  "properties": {
    "label": { },
    "minProperties": 0
} }
```

(f) JSC – PP 37714

Figure 12: Misplaced – bad nesting

```
{
  "type": "object",
  "oneOf": [ { "$ref": "#/definitions/..." } ],
  "definitions": {
    "geometry": {
      "description": "One geometry as defined by GeoJSON",
      "type": "object",
      "oneOf": [ {"$ref": "#/definitions/polygon" } ],
      "polygon": { }
} } }
```

Figure 13: Misplaced – ODS Definition Nesting

The extract in Figure 13 is of particular interest because of the 14,900 occurrences found in our corpus of OpenDataSoft generated schemas (two third of the corpus). Five object `definitions` (here only `"polygon"` is shown) are misplaced inside the `"geometry"` object and should really be moved one level up. As a result, the references in the internal `oneOf` combinator cannot find their targets.

5.3 Misnaming and Misspelling Issues

The set of 60 evolving keywords defined in JSON Schema are not well memorized by schema designers. A particular common confusion concerns extrema values for different elements which use 12 similar property names: `minLength` and `maxLength` with strings, `minimum`, `exclusiveMinimum`, `maximum` and `exclusiveMaximum` with numbers, `minItems`, `maxItems`, `minContains` and `maxContains` with arrays, `minProperties` and `maxProperties` with objects.

Figure 14 Extracts (a-d) illustrates typical min/max misnamings found in over 100 schemas in the corpus: Extract (a) should use `minLength` and `maxLength`. Extract (b) could be interpreted as a misplacement, the `minItems` belonging to the external array, but the overall context suggests that what was really meant is `minProperties` as a (strange) way to require at least one property. Extract (c) `minLength` should be `minItems` or is misplaced and should be one level down. Extract (d) is either misplaced, or should be `minLength`. Extracts (e-h) show more imaginary variants of min/max, from simple typos (`minitems`) to shorthands (`min`) and pure creations (`minValue minSize`). Incidentally, Extract (f) requires a number which must be a multiple of one…, thus the type could be simplified to `"integer"`.

```
{
  "date_of_birth": {
    "type": "string",
    "maximum": 10,
    "minimum": 10
} }
```

(a) JSC – PP 53086

```
{
  "type": "array",
  "items": {
    "type": "object",
    "minItems": 1
} }
```

(b) Store – Dependabot

```
{
  "type": "array",
  "items":
    { "type": "string" },
  "minLength": 1
}
```

(c) Store – Gitlab CI config

```
{
  "type": "object",
  "additionalProperties": {
    "type": "string",
    "minProperties": 1
} }
```

(d) JSC – PP 73935

```
{
  "type": "number",
  "min": 100,
  "max": 599,
  "integer": true
}
```

(e) JSC – PP 9882

```
{
  "type": "number",
  "multipleOf": 1,
  "minValue": 1
}
```

(f) JSC – PP 17072

```
{
  "type": "array",
  "items": { },
  "minSize": 1
}
```

(g) Store – Kite pipeline

```
{
  "type": "array",
  "minitems": "2",
  "maxitems": "2"
}
```

(h) JSC – PP 28955

Figure 14: Min/Max Multiple Confusions

```
{
  "type": "array",
  "additionalProperties": false,
  "items": { }
}
```

Figure 15: Misnamed – Store – XS App

In Figure 15, property `additionalProperties` is used on an array. Either it is a pure mistake, say a repetition of the previous pattern of definitions which were all objects, or `additionalItems` was meant.

The list of typographical errors is long, as they tend not to be detected because unknown keywords are ignored. The worst series is about meta-keyword `description` which occurs in our corpus

as *desription, despcription, descritpion, descrition, decription, descrp-tion, descripiton, descripition, decsription, descripion* and *Description*. Other variants are found for most keywords, such as **type:** (colon appended), **defaults**, **Default**, **refs**, **#refs**… There are also lots of mistakes around type names, with common errors such as: **null** instead of `"null"` or `"text"` instead of `"string"` (over 500 schemas each), but also `"int"`, `"list"` or `"bool"` used instead of `"integer"`, `"array"` or `"boolean"`. Another common pattern of type errors is due to **type**/**format** confusions, with predefined or other format names used as type names such as `tiny-int small-int medium-int uuid data-time`…

```
{
  "type": "string",
  "format": "url"
}
```

(a) Store – Jasonette

```
{
  "ref": "#/def...",
  "description": "..."
}
```

(b) Store – ESLint Config

```
{ "schema": "http://json-schema.org/draft-06/schema#" }
```

(c) Store – Unit Asm Def

```
{
  "description": "Unique ...",
  "readonly": true,
  "type": "int"
}
```

(d) JSC – PP 80381 13

```
{
  "title": "",
  "description": "",
  "type": null
}
```

(e) JSC – PP 54206

Figure 16: Misspelled keywords or format

Figure 16 illustrates misspelling issues on keywords and values: Extract (a) should use the `"uri"` format. Extracts (b, c) are both missing a **$** before **ref** and **schema**. Extract (d) uses an undefined type. Extract (e) **null** should be quoted.

5.4 Misversioning

Section 3.4 outlines the chaotic development of the JSON Schema specification. In this section, we analyze how well public schemas from our corpus conform to the various versions. We first investigate schemas globally, then separately schemas that advertise some version and schemas that do not advertise any version.

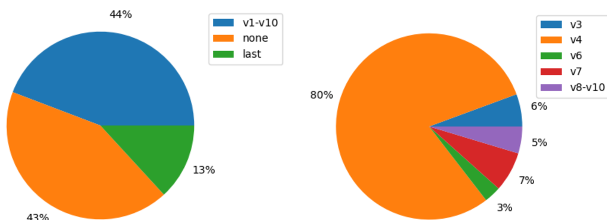

Figure 17: Share of versions in our corpus

Figure 17 outlines the share of JSON Schema versions in our corpus. On the left pie, 43% of schemas do not have any **$schema** (`none`), 44% advertise a schema with a specific number (`v1-v10`, v10 is `next`) and 13% advertise the latest possible schema (`last`, currently v9).

For schemas without any identified version, further analysis shows that they are mostly compatible with v4/v5 (98.5%) and v6/v7/v8 (96.0%). Only 35.5% are compatible with version 2020-12, mostly because of the change in how arrays are described.

The right pie shows the share of schema versions for schemas which advertise a specific version: overwhelmingly they use v4 (2013). Despite our efforts, we did not find many schemas from drafts 2019-09 and 2020-12. When a precise draft is advertised, 96% of schemas do conform to their version.

```
{
  "$schema": "http://json-schema.org/schema",
  "$id": "http://json.schemastore.org/cryproj.52.schema",
  "title": "CryProj schema",
  "$comment": "JSON Schema for CRYENGINE 5.2",
  "type": "object",
  "properties": { }
}
```

Figure 18: Implicit Latest Schema Version

The remainder 13% advertise implicitly the latest version with a non specific URL, as shown in Figure 18. The compatibility with the latest version is slightly reduced to 89% in this case, mostly because these schemas were developed for an earlier incompatible schema, usually v4.

```
{
  "$schema": "https://json-schema.org/draft-04/schema",
  "id": "...",
  "definitions": {
    "base_foo": {
      "$comment": "...",
      "id": "..."
} } }
```

(a) Ref – Draft 4 `ref` tests

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "https://json.schemastore.org/hws-config.json",
  "definitions": {
    "keyValuePairVariables": {
      "type": "object",
      "propertyNames": { "pattern": "^[A-Za-z0-9_.-]+$" }
} } }
```

(b) Store – HWS Config

Figure 19: Version Incompatibilities

Globally, over 400 schemas are not even loosely compatible with any version because of keywords pointing to different versions, as illustrated in Figure 19 with extracts which advertise a draft 4 version. Extract (a) mixes **id** which exists only until v5 with **$comment** which appears in v7. Although this error is harmless because comments do not have a semantics, its unintentional occurrence in JSON Schema own test suite demonstrates how easy it is to introduce mistakes in schemas, even for people involved in the specification process. Extract (b) also mixes the same keyword **id** with keyword **propertyNames** which appears only in v6.

5.5 Miscellaneous Issues

This section covers cases of missing intermediate keywords, mistaken (useless or invented) keywords, and mysterious patterns.

```
{
  "type": "array",
  "properties": {
    "namespace": { "type": "..." },
    "version": { "type": "..." }
  },
  "additionalProperties": false
}
```

(a) Store – SemGrep

```
{
  "type": "array",
  "anyOf": [
    { "type": "object" },
    { "type": "object" }
  ]
}
```

(b) Store – Bamboo

Figure 20: Missing Keywords

Extracts in Figure 20 are both missing a containing **items** to make sense of the object keyword associated to the array: in Extract (a) direct keywords **properties** and **additionalProperties**; in Extract (b) indirectly through keyword **anyOf**. The object-related keywords are ignored and the default permissive **items** applies, thus any data in the array is accepted.

```
{
  "description": "MicroTime",
  "format": "date-time",
  "type": "string",
  "additionalProperties": false
}
```

(a) Misc – K8S Microtime

```
{
  "type": "array",
  "items": { "$ref": "..." },
  "numItems": 2
}
```

(b) Store – Vega

Figure 21: Mistakes

In Figure 21, Extract (a) **additionalProperties** keyword is totally mistaken in an element which consists in a simple string.

Some schemas invent new keywords, not necessarily because they are using some extended vocabulary, but rather to express some semantics that could be available. Figure 14 (e) has an **integer** boolean property to suggest that the current element should be an integer. The **type** should really be set to "integer". Extract (b) in Figure 21 shows a non existing **numItems** keyword to fix the number of expected items in the array. It should use both **minItems** and **maxItems** for this purpose.

```
{
  "$defs": {
    "Foo": {
      "type": "object",
      "properties": { "foo": { "type": "string" } }
  } },
  "type": "object",
  "allOf": [ { "$ref": "#/$defs/Foo" } ],
  "properties": { "bla": { "type": "boolean" } }
}
```

Figure 22: *Of Mystery

A mystery, which is not a defect as such, is that 44.5% of all schemas in our corpus contain some **allOf**, **anyOf** or **oneOf** combinators with a list of only *one* schema which in most cases is a simple reference. Figure 22 outlines a typical usage pattern. A referenced object (Foo with property foo) is merged into the current object, so as to combine properties. This pattern raises two comments: (1) in many instances the combinator can be removed and the reference kept directly in the containing object, achieving the same result more simply; (2) this pattern defeats the very purpose of combinators, which should make clear what schemas are actually combined, whereas here the combination occurs with a schema which is actually *outside* of the combinator. We find this construction hard to understand: the structure would be clearer if both merged objects where *inside* a combinator. Although there is a clear use case for combining schemas as a building block for complex schemas, allowing such a pattern seems counter productive to good software engineering practices.

```
{
  "type": "number",  "minimum": 2,
  "enum": [ 2 ],     "maximum": 2
}
```

Figure 23: Store – Google Chrome

Another curiosity is shown in Figure 23, a lot of effort is spent on defining an integer constant with the help of 4 keywords. With recent versions this can be simplified to `{ "const": 2 }`.

5.6 Summary

Over 60% of schemas in our corpus have some kind of defect. The first cause of defect is a missing **type** declaration (52.8%). Most other defects are linked to nesting errors, which result in dangling references (26.7%, mostly from the OpenDataSoft generated schemas shown in Figure 13) or keywords which are ignored in the context because they are incompatible with the possible types (2.3%), possibly because of mistyping, misplacement, misnaming or mistakes. Other defects include schemas with a version mismatch (3.4% are inconsistent with their advertised schema, 0.7% do not have any compatible version) or various type errors, also due to misplacements, misnaming or misspelling. These defects leads us to suggest changes to JSON Schema.

6 RECOMMENDATIONS

The key takeaway from the previous section is that schemas are hard to write, and a lot of defects go unnoticed because of JSON Schema loose document semantics. Even in schemaless development frameworks allowed by scripting languages and flexible databases, this is paradoxical: if someone actually bothers to write a schema for their data, something they do not need for application features, then they probably want some help in checking data structures, which rather suggests an easy-to-use simple language with tight defaults. Based on Section 5 evidences, and in order to improve JSON Schema, we propose key changes that address these issues and could make the specification more useful.

A lot of defects are undetected because unknown or misplaced keywords are simply ignored. We thus recommend that it should not be the case, which means breaking the *keyword independence* feature and forbid unknown keywords, making schemas tight in effect. This can be achieved with the same overall syntax *if* keywords and values allowed at different levels are restricted, requiring:

- mandatory valid `$schema` version declaration at the root: if it does not claim to be a schema, it is not a schema;
- because they all imply a type, `type`, `const`, (type-homogeneous) `enum`, `$ref` and the three combinators `allOf anyOf oneOf` must be mutually exclusive inside an element; accepting any value is expressed with boolean schema `true`; consistently, the absence of an explicit or implicit type means *no* type, thus `{}` is `false` and should reject all values;
- type declarations must be simple scalars, not lists: type unions can be achieved with combinators `anyOf` and `oneOf`.
- type-specific keywords must only appear with an explicit compatible `type` at the same level; as a consequence, combinators only combine their list of schemas, without effects from *adjacent* keywords;
- unknown/unexpected keywords/values must generate errors;
- reference targets must exist: obviously the fact that reference must be resolved is already a constraint which we just reassert.
- formats must exist: formats for common types found in programming languages to cover very common use cases must be added.
- integers should just be integers, not floats that happen to be very close to an integer.

These restrictions result in possibly more verbose schemas because some description tricks cannot be used. However, because the constructs and types are directly exposed in the structure, we think that it is a positive improvement for readability and maintenance. Furthermore, well-scoped schemas help build validation tools which can compile subschemas without having to care about checks performed outside of the subschema specification.

These proposed changes amount to using a subset of JSON Schema with additional constraints. It raises two questions: What are the benefits? Are they practical? The actual key benefit is that it becomes harder to write invalid schemas with these constraints: the tight schema in Figure 2 conforms to these rules, whereas the loose schema in Figure 3 does not. Moreover, with these constraints (but ignoring the mandatory `$schema`), all but one defective schemas presented in Section 5 would be considered invalid: only Extract (e) in Figure 12 would pass as some object could indeed wish to forbid a property named `additionalProperties`. As a proof that these restrictions make sense in practice, we have tested them on 614 distinct samples from the *Store* source: 18% of these already conform to the restricted rules. This rate raises to 29% by simply imposing consistent `type` declarations with an `enum` instead of forbidding `type` in this case. If we ignore the 30% of schemas with errors in this corpus, it means that 42% of valid *Store* schemas are already conforming to the restricted rules. This suggests that many schemas are not far from being compatible with these restrictions.

The official test suite must be improved. Currently, only valid schemas are unit-tested against values, one keyword at a time, to check whether the semantics is correctly implemented. There is no test which looks like typical use cases found in actual schemas: for instance, most tests do not specify an explicit `type`, whereas most use cases have it. Although the current lax validation semantics makes it hard to generate an invalid schema, as far as we can tell not a single test in the official test suite has a dangling reference to check that it is detected. In order to help enforce these new restrictions, the official test suite must include invalid schemas which must be rejected by a conforming tool.

Section 5.3 shows that users struggle with JSON Schema many keywords. We propose to simplify the 60-keyword specification by removing seldom used features, especially if there is no strong theoretical argument to keep them:

- logical keywords `not if then else dependentRequired` should be reconsidered as they are seldom used, and a significant amount of use cases can be covered with combinators;
- special keywords `unevaluatedItems` and `unevaluatedProperties` are useless with the above restrictions because the extension semantics of combinators is simplified;
- little used array containment features `contains minContains maxContains` and the `multipleOf` constraint could also be removed;
- keywords `deprecated`, `readOnly` and `writeOnly` do not seem to belong to a data structure description, but rather to an API description: as they are off-topic, they should be removed;
- features `dynamicAnchor` and `dynamicRef` should be removed because of the implied complexity induced on the validation algorithm [7];
- content keywords `content*` could be managed by relying on `format`, e.g. `"text/html; charset=UTF-8"`;
- most Keyword `propertyNames` use cases are covered with `patternProperties`, so it could be removed;
- Keyword `$vocabulary` usage beyond JSON Schema own meta-schema seems very scarce, extensions could rather be managed with external `$ref` pointing to specific URI;
- keywords `minimum` and `maximum` could be replaced by `minValue` and `maxValue` for homogeneity with other extrema keywords.

Overall, we propose to remove about 20 keywords. One feature that could be added is a combinator which merges properties from objects to define a new object, as discussed in the next section.

With the above restrictions and simplifications, a schema definition would be tight. This proposal goes beyond currently discussed changes [17]. We also think that JSON Schema should go a step further so that schemas should be tight by default and that it should require some minimal effort, i.e. additional keywords, to loosen them. This would imply changing the philosophy, such as an `optional` keyword to replace `required`, and using `false` as a default for `additionalProperties` and other keywords.

7 JSON MODEL OUTLINE

Although the recommendations above should reduce the number of defects found in JSON schemas, the architecture of the language still results in verbose descriptions. JSound [1] is a alternative data description language proposed for the same purpose, which is also quite verbose and less expressive than JSON Schema. Its compact form is much more dense, as expected, but also lacks some features. This section briefly outlines JSON Model [13], a compact and expressive alternative for describing JSON data structures.

The first principle of JSON Model is to rely as much as possible on type inference to represent data structures, so that the simplest values (`null true 0 0.0 ""`) represent their types, and to resort to more complex object descriptions only when necessary. This is

```
{
  "name": "",
  "age": 0,
  "?friends": [ "" ]
}
```

(a) Tight Model for Figure 1

```
{
  "^[A-Z]$": "_",
  "$DATE": "=42",
  "": "$U64"
}
```

(b) Constants and Types

Figure 24: Simple JSON Model Examples

illustrated in Figure 24 (a): the model describes an object with 3 properties. The two first properties are mandatory. The first expects a string, the second an integer. Property `friends` expects an array of strings and is optional because the property name starts with a **?**. Models are tight by default, i.e. unexpected properties are rejected. Figure 24 (b) shows an object definition with constants (**=**), regular expressions (**^**) and and predefined types (**$**): uppercase one-letter property names (written with a regex) expect an empty string value, **_** is the string escape character; property names which match a date expect a constant integer 42 value; any other property name expects an unsigned 64-bit integer value.

```
{
 "@": [ "" ],
 "eq": 7,
 "distinct": true
}
```

(a) Constraints

```
{ "|": [
    { "tag": "=1", "isbn": "$ISBN" },
    { "tag": "=2", "issn": "$ISSN" }
  ]
}
```

(b) Composition

Figure 25: JSON Model Constraints and Composition

Further constructs allow to build more advanced types. Figure 25 (a) is a constrained model: the expected target type `"@"` is an array of 7 unique strings. Figure 25 (b) is a model composition with an *or* operator `"|"`: depending on the tag integer value, the object must have a mandatory `isbn` or `issn` property with its associated type.

```
{
  "%": { "geo": "$https://json-model.org/geom" },
  "polygons": [ "$geo#polygon" ]
}
```

Figure 26: Model Loading and Referencing

Figure 26 introduces referencing existing models and definitions through URLs: `geo` loads a model through a URL, and the `polygon` type defined in the loaded model is used for array items.

Finally, Figure 27 illustrates advanced features: the model includes a comment **#**, two type definitions (`Person` and `Structure`) inside **%**, with a recursion on `Structure` as it depends on itself, and the `"+"` operator which allows to *merge* properties of object definitions, so that string Property `company` is only required at the root element. Figure 28 is a valid sample value for this model.

JSON Model is a work in progress. The description language currently contains 19 keywords or sentinel characters. Open questions remain: instead of letter-based mnemonics (`le eq distinct`...), constraint properties could use mathematical symbols less likely to be used in typical data structure; special values could be used

```
{
  "#": "Organizational Chart",
  "%": {
    "Person": { "name": "", "?title": "" },
    "Structure": {
      "head": "$Person",
      "?subs": [ "$Structure" ]
    }
  },
  "+": [ { "company": "" }, "$Structure" ]
}
```

Figure 27: Recursive *Organizational Chart* Model

```
{
  "company": "Jason Co",
  "head": { "title": "CEO", "name": "Albertine" },
  "subs": [
    { "head": { "title": "CFO", "name": "Betty" } },
    {
      "head": { "title": "CTO", "name": "Carol" },
      "subs": [
        { "head": { "title": "Prod Lead", "name": "Dany" } },
        { "head": { "title": "Dev Lead", "name": "Elen" } }
      ]
    }
  ]
}
```

Figure 28: Sample Value for Figure 27

for implicit constraints on numbers, *e.g.* `-1` for any integer, `0` for positive and `1` for strictly positive; regular expressions could use a `/` sentinel like some other languages...

As JSON Model is written in JSON, there is a self-validating tight meta-model. We think that the design is intelligible, as above model illustrations can be broadly understood by a developer without prior knowledge of the formal syntax. A proof-of-concept implementation written in Python is available [12]. It offers both a validator (interpreter), a compiler, and partial schema-to-model and model-to-schema converters.

8 CONCLUSION

In this paper, we have brought 6 main contributions:

(1) a comprehensive study of the state of the JSON Schema specification and its evolution, with an emphasis on error-prone features (Section 3);
(2) an extensive JSON Schema corpus aggregated from previous works and complemented with our own collection of schemas, available online [30] and reaching over 57,800 distinct public schemas (Section 4);
(3) analysis tools for processing schemas and looking for typical defects, available online [13, 14] (Section 4);
(4) a categorization of defects based on their causes (Section 5);
(5) propositions to improve the JSON Schema specification so as to avoid most of these defects (Section 6);
(6) the outline of JSON Model, an compact and expressive JSON-based data-structure description language (Section 7).

An obvious limit to our study is the data collection. Despite our efforts, we found very little samples of schemas over v8, our corpus is heavily slanted towards v4 and v7. A lot of tool implementations seem stuck to v4. MongoDB schemas are described in BSON with an extension of JSON Schema v4; this is also the case of several Postgres implementations such as `is_jsonb_valid` (which is actively maintained) and `postgres-json-schema` (which is not). Pydantic also seems to generate v4-compatible schemas. A lot of recent samples only come from the official JSON Schema Test Suite. These unit-test oriented samples are quite far from realistic usage patterns.

A second limit is our static analysis tools: we only find what we are looking for, thus if some errors are not actively investigated they will just be missed. Moreover, our implementation is not perfect, for instance it does not resolve external references and includes a few shortcuts to deal with distinct versions of JSON Schema in one go, so a few false positives or negatives reports are still possible.

Checking data structures at API interfaces, a typical use case for JSON Schema, is highly biased: in the development phase, schemas are useful as a documentation and can help debugging by catching malformed data; even if a schema is too loose, malformed data are not likely to fall specifically on the loose elements, and anyway such errors will be detected with functional tests; once in production, data generated by the application code will very probably be conforming, thus schema validations will nearly always return true. Keeping a data validation is a loss of cpu time and energy: one reason to keep it anyway is to prevent malicious clients from inserting invalid data as a mean of cyberattack; another is to provide data quality assurances, in particular when an interface is open to external unchecked clients.

Another bias, typical of open data platforms, is that the purpose of schemas seems much more about documenting data formats than actually being used to validate data. In such a context, the readability of the schema should be paramount. Given the complexity of JSON Schema, this tool is not well suited for this purpose, and other formats such as YAML are more appropriate.

We think that JSON Schema is not a good fit to the two purposes it has been created for: as a data structure development tool, it is too verbose and error-prone; as a documentation tool, the JSON format is not ideal for human consumption and JSON Schema complexity makes it hard to understand. The changes we propose should help the data-structure side, but the result, even with fewer keywords and a more rigid structure, is still quite verbose. This motivates our proposed lightweight alternative (Section 7).

Finally this work also raises questions about Standard governance and how evidences and feedbacks are collected to try to improve a proposal. Overall, it seems that JSON Schema has gone towards more and more complex semantics, whereas simplicity is probably what is really needed.

We would like to thank Olivier Hermant and the anonymous reviewers for their help in proofreading this paper.

REFERENCES

[1] Cesar Andrei, Daniela Florescu, Ghislain Fourny, Jonathan Robie, and Pavel Velikhov. 2018. JSound 2.0 – The Complete Reference. https://jsound-spec.org

[2] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. Witness Generation for JSON Schema. *Proceedings of the VLDB Endowment (PVLDB)* 15, 13 (Sept. 2022), 4002–4014.

[3] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2023. Validation of Modern JSON Schema: Formalization and Complexity. (March 2023). working paper or preprint.

[4] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. A Type System for Interactive JSON Schema Inference (Extended Abstract). In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 132)*. Schloß Dagstuhl–Leibniz-Zentrum für Informatik, 101:1–101:13.

[5] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. A JSON Schema Corpus. https://github.com/sdbs-uni-p/json-schema-corpus.

[6] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents. In *40th Int. Conf. on Conceptual Modeling ER 2021 (Lecture Notes in Computer Science, Vol. 13011)*. Springer, 102–112.

[7] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. *Negation-Closure for JSON Schema*. Preprint. https://arxiv.org/abs/2202.13434v1

[8] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conf. on Object-Oriented Programming*. Springer, 257–281.

[9] Pierre Bourhis, Juan L Reutter, Fernando Suárez, and Domagoj Vrgoč. 2017. JSON: Data model, Query languages and Schema specification. In *PODS 2017 – Proceedings of the Thirty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. Chicago, USA.

[10] Byron Choi. 2002. What are read DTDs like?. In *Fifth Int. Workshop on the Web and Databases (WebDB)*. 43–48.

[11] Fabien Coelho, Alexandre Aillos, Samuel Pilot, and Shamil Valeev. 2011. A Field Analysis of Relational Database Schemas in Open-source Software. In *3rd Int. Conf. on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, IARIA (Ed.). 9–15.

[12] Fabien Coelho and Claire Yannou-Medrala. 2023. JSON Model. https://github.com/clairey-zx81/json-model

[13] Fabien Coelho and Claire Yannou-Medrala. 2023. *JSON Model: a Lightweight Featureful Description Language for JSON Data Structures*. Tech. Report A/795/CRI. CRI, Mines Paris – PSL.

[14] Fabien Coelho and Claire Yannou-Medrala. 2023. JSON Schema Statistics Tools. https://github.com/clairey-zx81/json-schema-stats

[15] Hugo André Coelho Cardoso and José Carlos Ramalho. 2022. Synthetic Data Generation from JSON Schemas. In *11th Symposiumn on Languages, Applications and Technologies (SLATE)*.

[16] Douglas Crockford. 2006. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627. IETF.

[17] Greg Dennis. 2023. The Last Breaking Change. Blog Post. https://json-schema.org/blog/posts/the-last-breaking-change

[18] ECMA International. 2011. *Standard ECMA-262 – ECMAScript Language Specification* (5.1 ed.). 846 pages. First edition in 1999.

[19] 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 *39th International Conference on Conceptual Modeling ER (Workshops) 2020 (LNCS, Vol. 12584)*. Springer, Vienna, Austria, 220–230.

[20] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2019. Type Safety with JSON Subschema. *CoRR* abs/1911.12651 (2019). arXiv:1911.12651

[21] JSON Schema. 2023. Implementations. (2023). https://json-schema.org/implementations.html.

[22] Benjamin Maiwald, Benjamin Riedle, and Stefanie Scherzinger. 2019. What Are Real JSON Schemas Like? – An Empirical Analysis of Structural Properties. In *Advances in Conceptual Modeling - ER 2019 Workshops FAIR, MREBA, EmpER, MoBiD, OntoCom, and ER Doctoral Symposium Papers (LNCS, Vol. 11787)*. Springer, Salvador, Brazil, 95–105.

[23] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. 2003. The XML Web: a First Study. In *The Web Conference*.

[24] Mitre. 2021. *Inefficient Regular Expression Complexity*. CWE 1333.

[25] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON schema. In *Proceedings of the 25th international conference on World Wide Web*. 263–273.

[26] Stefanie Scherzinger and Sebastian Sidortschuck. 2020. An Empirical Study on the Design and Evolution of NoSQL Database Schemas. LNCS 12400, Springer, 441–455.

[27] Sideways Inc. 2023. *joi – Schema Description Language and Data Validator for JavaScript*. Web Site. https://joi.dev First version in 2013.

[28] Lanju Wang, Oktie Hassanzadeh, Shuo Zhang, Juwei Shi, Limei Jiao, Jia Zou, and Chen Wang. 2015. Schema Management for Document Stores. In *VLDB Endowment*, Vol. 8. 922–933.

[29] Austin Wright, Henry Andrews, Ben Hutton, and Greg Dennis. 2022. *JSON Schema: A Media Type for Describing JSON Documents*. Draft 2020-12. IETF.

[30] Claire Yannou-Medrala and Fabien Coelho. 2023. Yet Another JSON Schema Corpus. https://github.com/clairey-zx81/yac