Finding Data Compatibility Bugs with JSON Subschema Checking

Andrew Habib[*]
SnT, University of Luxembourg
Luxembourg
andrew.a.habib@gmail.com

Avraham Shinnar
Martin Hirzel
IBM Research
USA
shinnar,hirzel@us.ibm.com

Michael Pradel
University of Stuttgart
Germany
michael@binaervarianz.de

ABSTRACT

JSON is a data format used pervasively in web APIs, cloud computing, NoSQL databases, and increasingly also machine learning. To ensure that JSON data is compatible with an application, one can define a JSON schema and use a validator to check data against the schema. However, because validation can happen only once concrete data occurs during an execution, it may detect *data compatibility bugs* too late or not at all. Examples include evolving the schema for a web API, which may unexpectedly break client applications, or accidentally running a machine learning pipeline on incorrect data. This paper presents a novel way of detecting a class of data compatibility bugs via *JSON subschema checking*. Subschema checks find bugs before concrete JSON data is available and across all possible data specified by a schema. For example, one can check if evolving a schema would break API clients or if two components of a machine learning pipeline have incompatible expectations about data. Deciding whether one JSON schema is a subschema of another is non-trivial because the JSON Schema specification language is rich. Our key insight to address this challenge is to first reduce the richness of schemas by canonicalizing and simplifying them, and to then reason about the subschema question on simpler schema fragments using type-specific checkers. We apply our subschema checker to thousands of real-world schemas from different domains. In all experiments, the approach is correct whenever it gives an answer (100% precision and correctness), which is the case for most schema pairs (93.5% recall), clearly outperforming the state-of-the-art tool. Moreover, the approach reveals 43 previously unknown bugs in popular software, most of which have already been fixed, showing that JSON subschema checking helps finding data compatibility bugs early.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Software evolution**; *Software version control*.

KEYWORDS

JSON schema, data compatibility bugs, subschema checking

ACM Reference Format:
Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding Data Compatibility Bugs with JSON Subschema Checking. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464796

1 INTRODUCTION

JSON (JavaScript Object Notation) is a data serialization format that is widely adopted to store data on disk or send it over the network. The format supports primitive data types, such as strings, numbers, and Booleans, and two possibly nested data structures: arrays, which represent ordered lists of values, and objects, which represent unordered maps of key-value pairs. JSON is used in numerous applications. It is the most popular data exchange format in web APIs, ahead of XML [51]. Cloud-hosted applications also use JSON pervasively, e.g., in micro-services that communicate via JSON data [43]. On the data storage side, not only do traditional database management systems, such as Oracle, IBM DB2, MySQL, and PostgreSQL, now support JSON, but two of the most widely deployed NoSQL database management systems, MongoDB and CouchDB/Cloudant, are entirely based on JSON [50]. Beyond these applications, JSON is also gaining adoption in machine learning [21, 54].

With the broad adoption of JSON as a data serialization format soon emerged the need for a way to describe how a JSON document should look. For example, a web API that consumes JSON data can avoid unexpected behavior if it knows the structure of the data it receives. *JSON Schema* declaratively defines the structure of nested values (JSON documents) via types (JSON schemas) [46]. A JSON Schema *validator* checks whether a JSON document d conforms to a schema s. JSON Schema validators exist for many programming languages and are widely used to make software more reliable [58].

Despite the availability of JSON schema validators, some data-related bugs may get exposed late in the development process or even remain unnoticed until runtime misbehavior is observed. As one example, consider a RESTful web API for which the data types are specified with JSON schemas. If the API, and hence the schemas, evolve, the revised schemas may not be backward compatible and unexpectedly break client applications [29, 42]. As another example, consider a machine learning pipeline where the data, as well as the input and output of operations, are specified with JSON schemas [21, 54]. If some data is incorrect or different components of the pipeline have incompatible expectations, the pipeline may

[*] Most of the work was conducted while at IBM Research and at TU Darmstadt.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464796

compute incorrect results or crash after hours of computation. For both of the above scenarios, schema validators can detect these problems only at runtime, because that is when concrete JSON data is available for validation. Even worse, the problems may remain unnoticed until some data that triggers the problem occurs.

We call such problems *data compatibility bugs*, which means that two pieces of software that share some JSON data have incompatible expectations about the data. To estimate the prevalence of JSON schemas and the potential for bugs caused by them, we perform an initial analysis of 25 GitHub projects with at least one schema (sampled from thousands of such projects). These projects contain 3,806 schema files, 737 of which have been changed at least once. This shows that JSON schemas are common and also commonly change, potentially introducing bugs. Moreover, this work is motivated by JSON schemas, and bugs related to them, within a major industrial player. We see a clear need for detecting this class of bugs.

This paper presents a novel way of detecting data compatibility bugs early. The key idea is to check for two given JSON schemas whether one schema is a subschema (subtype) of the other schema. Because such a check can be performed on the schema-level, i.e., independently of concrete JSON data, the approach can detect data compatibility bugs earlier and more reliably than JSON schema validation alone. For the first of the above examples, our approach can check whether a revised schema is a subtype or a supertype of the original schema. If it is neither, then the schema evolution is a breaking API change that should be communicated accordingly to clients. For the second example, the approach can check if the schema of the input given to an ML pipeline step is a subtype of the schema expected by that step, which reveals bugs before running the pipeline. Our empirical evaluation finds real-world bugs related to both kinds of problems.

Deciding whether one JSON schema is a subtype of another is far from trivial because JSON Schema is a surprisingly complex language. Semantically equivalent schemas can look syntactically dissimilar. Schemas for primitive types involve sophisticated features, such as regular expressions for strings, that interact with other features, such as string length constraints. JSON Schema supports enumerations (singleton types) and logic connectives (conjunction, disjunction, and negation) between schemas of heterogeneous types. Even schemas without explicit logic connectives often have implicit conjunctions and disjunctions. JSON schemas or nested fragments thereof can be uninhabited. As a result of these and other features of JSON Schema, simple structural comparison of schemas is insufficient to answer the subschema question.

Our work addresses these challenges based on the insight that the subschema problem can be decomposed into simpler subproblems. Given two JSON schemas, the approach first canonicalizes and then simplifies the schemas using a series of transformations. While preserving the semantics of the schemas, these transformations reduce the number of cases and ensure that the schemas consist of schema fragments that each describe a single basic type of JSON data. This homogeneity enables the last step, recursively performing the subtype check using type-specific checkers. We have built an open-source tool, *jsonsubschema*, that always terminates, returning one of three answers for a subschema check: true, false, or unknown. It returns unknown for a small set of rarely occurring features of schemas. When it returns true or false, it is designed to be always correct, and in our experiments, this was indeed the case.

The most closely related prior work is an open-source project called *is-json-schema-subset (issubset)* [35]. It only handles a fraction of the features of JSON Schema and, as we show experimentally, often gives incorrect answers. The problem of subschema checking has also been explored for XML [23, 39], where it is called schema containment [56]. However, that approach treats XML schemas as tree automata, which has been shown to be insufficient for JSON schemas because it is more expressive than tree automata [46].

We evaluate the approach with thousands of real-world JSON schemas gathered from different domains, including schemas used for specifying web APIs, cloud computing, and machine learning pipelines. In our evaluation, the approach was 100% precise and 100% correct, and it successfully decided the subschema question for most schema pairs, with a recall of 93.5%. Our approach clearly outperforms the closest existing tool in terms of all three metrics. Applying JSON subschema checking for bug detection, we find 43 data compatibility bugs related to API evolution and mistakes in machine learning pipelines. All bugs are confirmed by developers and 38 are already fixed. Developers acknowledged our work and emphasized the severity of many of the bugs.

In summary, this paper makes the following contributions:

- Formulating the problem of detecting data compatibility bugs as JSON subschema checking (Section 2).
- A canonicalizer and simplifier that converts a given schema into a schema that is simpler to check yet permits the same set of documents (Sections 3.1 and 3.2).
- A subschema checker for canonicalized JSON schemas that uses separate subschema checking rules for each basic JSON type (Section 3.3).
- Empirical evidence that the approach outperforms the state of the art, and that it reveals real-world data compatibility bugs in different domains (Section 5).

Our tool is open-source at https://github.com/ibm/jsonsubschema.

2 PROBLEM STATEMENT

2.1 Background

JSON Schema is a declarative language for defining the structure and permitted values of a JSON document [58]. This work focuses on JSON Schema draft-04 [31], one of the most widely adopted versions. JSON Schema itself uses JSON syntax. To specify which data types are allowed, JSON Schema uses the keyword `type` either with one type name (e.g., `{'type':'string'}`) or with a list of type names (e.g., `{'type':['null','boolean']}`). Each JSON type has a set of validation *keywords* that restrict the values a schema of this type permits. For example, `{'type':'integer','minimum':0}` restricts integers to be non-negative, whereas `{'type':'string','pattern':'^[A-Za-z0-9]+$'}` restricts strings to be alphanumeric.

In addition to type-specific keywords, JSON Schema allows enumerating exact values with `enum` and combining different schemas using logic connectives. For example, schema `{'enum':['a', [], 1]}` restricts the set of permitted JSON values to the string literal `'a'`, an empty array, or the integer 1. Logic connectives, such as `anyOf`, `allOf`, and `not`, allow schema writers to express disjunctions, conjunctions, and negations of schemas. The empty schema, `{}`, is the

```
schema      ::= {type?, strKw, numKw, arrKw, objKw,
                 enum?, not?, allOf?, anyOf?, oneOf?, ref?}
type        ::= 'type': (typeName | [ typeName+ ])
typeName    ::= 'null' | 'boolean' | 'string' | 'number' |
                'integer' | 'array' | 'object'
strKw       ::= minLength?, maxLength?, pattern?
minLength   ::= 'minLength': NUM
maxLength   ::= 'maxLength': NUM
pattern     ::= 'pattern': REGEX
numKw       ::= minimum?, maximum?, exclMin?, exclMax?,
                multOf?
minimum     ::= 'minimum': NUM
maximum     ::= 'maximum': NUM
exclMin     ::= 'exclusiveMinimum': BOOL
exclMax     ::= 'exclusiveMaximum': BOOL
multOf      ::= 'multipleOf': NUM
arrKw       ::= items?, minItems?, maxItems?,
                addItems?, uniqItems?, contains?
items       ::= 'items': (schema | [ schema+ ])
minItems    ::= 'minItems': NUM
maxItems    ::= 'maxItems': NUM
addItems    ::= 'additionalItems': (BOOL | schema)
uniqItems   ::= 'uniqueItems': BOOL
contains    ::= 'contains': schema
objKw       ::= props?, minProps?, maxProps?, required?,
                addProps?, patProps?, depend?
props       ::= 'properties': {(STR : schema)* }
minProps    ::= 'minProperties': NUM
maxProps    ::= 'maxProperties': NUM
required    ::= 'required': [STR* ]
addProps    ::= 'additionalProperties': (BOOL|schema)
patProps    ::= 'patternProperties': {
                  (REGEX : schema)* }
depend      ::= 'dependencies': {
                  (STR : (schema | [ STR+ ]))* }
enum        ::= 'enum': [ VALUE+ ]
not         ::= 'not': schema
allOf       ::= 'allOf': [ schema+ ]
anyOf       ::= 'anyOf': [ schema+ ]
oneOf       ::= 'oneOf': [ schema+ ]
ref         ::= '$ref': PATH
```

Figure 1: Grammar of full JSON Schema.

top of the schema hierarchy, i.e., all documents are valid for `{}`. The negation of the empty schema, `{'not':{}}`, is the bottom of the hierarchy, i.e., no documents are valid for it. Finally, the keyword `$ref` retrieves schemas using URIs and JSON pointers. JSON validation against a schema with `$ref` has to satisfy the schema retrieved from the specified URI or JSON pointer. We refer the interested reader to the full specification of JSON Schema [31] and its formalization [46].

Figure 1 shows the grammar for JSON schemas. The start symbol is *schema*. A schema can mix keywords for all types as well as logic connectives. All-caps indicates literal tokens such as *NUM* or *BOOL* whose lexical syntax follows the usual conventions of JSON. Some keywords can be specified in multiple ways. For instance, *type* can be just one type name or a list of types; *items* can be just one schema or a list of schemas; and *addItems* can be a Boolean or a schema.

2.2 JSON Subschema Problem

This paper presents how to detect data compatibility bugs by addressing the JSON subschema problem. Suppose a schema validator that determines whether a JSON document d is valid according to a schema s: $valid(d, s) \rightarrow \{True, False\}$. A schema s is a *subschema* of another schema t, denoted $s <: t$, if and only if $\forall d : valid(d, s) \implies valid(d, t)$. The subschema relation is a form of subtyping that views a type (schema) as a set of values (JSON documents) [45].

As an example, consider an excerpt of versions 0.6.1 and 0.6.2 of a real-world schema that describes an API from a collection of schemas for content used by the Washington Post [13]:

Version 0.6.1:

```
{'type': 'object',
 'properties': {
   'category': {
     'type': 'string',
     'enum': ['staff', 'wires',
          'other']}}}
```

Version 0.6.2:

```
{'type': 'object',
 'properties': {
   'category': {
     'type': 'string',
     'enum': ['staff', 'wires',
          'stock', 'other']}}}
```

Both schemas describe an object with a property "category" with a fixed set of values. Version 0.6.1 is a subschema of version 0.6.2 because all documents valid according to the first are also valid according to the latter. In contrast, version 0.6.2 is not a subschema of version 0.6.1 because the JSON document `{'category':'stock'}` is valid in version 0.6.2 but not in version 0.6.1. To retain backward compatibility, this evolution is fine for API arguments, but may break clients if the schema describes an API response.

2.3 Challenges

The rich feature set of JSON Schema makes establishing or refuting a subtype relation between two schemas non-trivial. Even for simple, structurally similar pairs of schemas, such as `{'enum':[1, 2]}` and `{'enum':[2, 1]}`, equivalence does not hold through textual equality. There are several challenges for algorithmically checking the JSON schema subtype relation.

First, the same set of JSON values, i.e., the same type, can be described in several different syntactical forms, i.e., schemas. For example, Figure 2 shows five equivalent schemas describing a JSON value that is either a non-empty string or null.

Second, even for primitive types, such as strings and numbers, nominal subtyping is not applicable. JSON Schema allows various constraints on primitive types, resulting in non-trivial interactions not captured by nominal types. For example, inferring that an `integer` schema is a subtype of a `number` requires properly comparing the range and multiplicity constraints of the schemas.

Third, logic connectives combine non-homogeneous types, e.g., `'string'` and `'null'` in Figure 2b. Moreover, enumerations restrict types to predefined values, which require careful handling, especially when enumerations interact with non-enumerative types, such as in Figures 2a, 2b, and 2c.

Fourth, the schema language allows implicit conjunctions and disjunctions. For example, Figure 2b has an implicit top-level conjunction between the subschemas under `anyOf` and `not`. As another example, a schema that lacks a `type` keyword, such as the schema

```
{'type': ['null', 'string'],
  'not': {'enum': ['']}}
```

(a)

```
{'anyOf': [{'type': 'null'},
           {'type': 'string'}],
 'not': {'type': 'string',
         'enum': ['']}}
```

(b)

```
{'allOf': [{'anyOf': [
              {'type': 'null'},
              {'type': 'string'}]},
  {'not': {'type': 'string', 'enum': ['']}}]}
```

(c)

```
{'anyOf': [ {'type': 'null'},
  {'type': 'string', 'pattern': '.+'}]}
```

(d)

```
{'allOf': [
    {'anyOf': [
        {'type': 'null'},
        {'type': 'string'}]},
    {'anyOf': [
        {'type': 'boolean'}, {'type': 'null'},
        {'type': 'number'}, {'type': 'integer'},
        {'type': 'array'}, {'type': 'object'},
        {'type': 'string', 'pattern': '.+'}]}]}
```

(e)

Figure 2: Five syntactically different but semantically equivalent schemas for a value that is either a non-empty string or null.

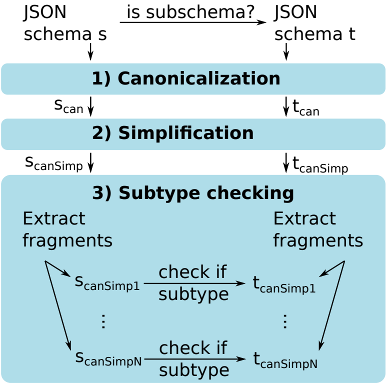


Figure 3: Overview of JSON subschema checker.

{'pattern':'.+'}, has an implicit disjunction of all possible types, while still enforcing any type-specific keyword, such as the pattern for strings only. Figure 2e makes this implicit disjunction explicit.

Finally, JSON Schema also allows uninhabited types. That is, a schema can be syntactically valid yet semantically self-contradicting, e.g., {'type':'number','minimum':5,'maximum':0}. Such schemas validate no JSON value at all and complicate reasoning about subtyping.

3 APPROACH

This section describes how we address the problem of checking whether one JSON schema is a subtype of another. Given a pair of JSON schemas, the approach returns either true, false, or unknown. The first two answers are guaranteed to be correct (assuming no bugs in our algorithm or implementation), while the third occurs rarely in practice. Because JSON schemas are complex, creating a subtype checker for arbitrary schemas directly would necessitate a complex algorithm. A key insight of our work is to instead decompose the problem into three steps, outlined in Figure 3. The first step canonicalizes a schema into an equivalent but more standardized schema (Section 3.1). The second step further simplifies a schema, eliminating enumerations, negation, intersection, and schema unions where possible (Section 3.2). Finally, the two canonicalized and simplified schemas are compared by extracting and comparing type-homogeneous schema fragments (Section 3.3).

The canonicalization and simplification steps reduce the ways the different features of JSON Schema are used, and completely eliminate some features. Table 1 summarizes the properties that schemas have after canonicalization and after simplification. The core principle is to convert a schema to an equivalent but more tractable form. Equivalent means that the schema still accepts the same set of documents. More tractable means that the schema uses fewer and less entangled keywords with fewer forms. We next describe these properties in more detail and explain how we ensure them. For readers interested in a more formal description, [33] describes the canonicalization and simplification as rewriting rules.

3.1 Canonicalizing JSON Schemas

This section introduces a canonicalization procedure that transforms any JSON schema into an equivalent canonical schema. Canonicalization enforces two main properties. JSON Schema allows schemas to mix specifications of different types. To enable local, domain-specific reasoning in the subtype checker, canonicalization first splits up these schemas into smaller, homogeneously typed schemas combined with logic connectives. JSON Schema also allows many alternative ways to represent the same thing. Additionally, most keywords can be omitted and defaults assumed. Canonicalization picks, when possible, one form, and explicates omitted defaults.

To ensure these properties, canonicalization applies a series of transformations, which each takes a schema and rewrites it into a semantically equivalent but more canonical schema. The schema is transformed until no more transformations are applicable, yielding a canonical schema. The following describes the transformations, starting with type-independent transformations, followed by transformations addressing the different language features in Table 1.

Type-independent transformations. To enable reasoning about one type at a time, we transform a schema whose type is a list, such as in the example in Figure 2a, by making the implicit disjunction explicit using anyOf, as shown in Figure 2b. Moreover, we transform schemas that contain a logical connective mixed with other properties (e.g., other logical connectives), such as in the example in Figure 2b. By making the implicit conjunction explicit using allOf, we isolate logical connectives, as shown in Figure 2c.

Eliminating Booleans and integers. We eliminate these types by representing them with other features of JSON Schema: boolean schemas become an enumeration of true and false, and integer schemas become number schemas constrained to be a multiple of 1.

Representing strings as patterns. In full JSON Schema, strings may be restricted based on their minimum length, maximum length, and a regular expression. To reduce the number of cases, and since length constraints interact with the pattern, if specified, minLength and maxLength keywords are transformed into a semantically equivalent regular expression (intersected with the pattern schema), so canonicalized string schemas only have the keyword pattern.

Canonicalizing arrays. We canonicalize schemas that describe arrays using two transformations that reduce the number of ways

Table 1: Properties of canonicalized (Section 3.1) and simplified (Section 3.2) schemas.

Language	Use of feature in schemas		
feature	**Full JSON Schema**	**Canonicalized**	**Simplified**
`null`	(no keywords)	Yes	Yes
`boolean`	(no keywords)	Represented as `enum`	Represented as `enum`
`string`	`{min,max}Length, pattern`	Keyword `pattern` only	Keyword `pattern` only
`number`	`{min,max}imum, multipleOf, exclusive{Min,Max}imum`	All keywords	All keywords
`integer`	(same keywords as number)	Eliminated	Eliminated
`array`	`{min,max}Items, items, additionalItems, uniqueItems`	All keywords, but `items` is always a list and `additionalItems` is always a schema	All keywords, but `items` is always a list and `additionalItems` is always a schema
`object`	`properties, {min,max}Properties, required, additionalProperties, patternProperties, dependencies`	Only keywords `{min,max}Properties, required, patternProperties`	Only keywords `{min,max}Properties, required, patternProperties`
`enum`	Heterogeneous, any type	Homogeneous, any type	Only for `boolean`
`not`	Mixed connectives	Isolated connective	Only for `number`, `array`, `object`
`allOf`	Mixed connectives	Isolated connective	Only for `not`
`anyOf`	Mixed connectives	Isolated connective	Only for `not`, `allOf`, `array`, `object`, and disjoint `number`
`oneOf`	Mixed connectives	Isolated connective	Eliminated

in which the `items` and `additionalItems` keywords may be used. With `items`, a schema can define what data may be stored in an array. With `additionalItems`, a schema can define any data in an array that is not specified in `items`. The first transformation handles the fact that the `items` keyword may hold either a list of schemas, which restricts the respective items in the array, or a single schema, which restricts all items in the array in the same way. The transformation eliminates the second case by using the `additionalItems` keyword, e.g.:

```
{'type': 'array',                    {'type': 'array',
 'items': {                      →    'additionalItems': {
   'type': 'number'}}                   'type': 'number'}}
```

Since `additionalItems` may be either a schema or a Boolean (`false` disallows additional items), the second transformation replaces a Boolean `additionalItems` with a corresponding JSON schema, where the schemas `{}` and `{'not':{}}` replace `true` and `false`, respectively. So `additionalItems` becomes always a schema.

Canonicalizing objects. Schemas for objects have various keywords. To reduce complexity, our approach eliminates the keywords `properties`, `additionalProperties`, and `dependencies` by rewriting them into `required` and `patternProperties`. Moreover, canonicalization ensures that `patternProperties` uses non-overlapping regular expressions. For example, this object schema would be canonicalized as follows:

```
{'type': 'object',                      {'type': 'object',
 'properties': {                         'patternProperties': {
   'a': {'type': 'string'},                '^a$': {'type': 'string'},
   'b': {'type': 'array'}}         →       '^b$': {'type': 'array'},
 'patternProperties': {                    '([^a]+a|a.).*': {
   'a': {'type': 'boolean'}}}                'type': 'boolean'}}}
```

The non-canonical schema on the left describes the types of properties “a” and “b” using `properties`, and of any property that contains an “a” using `additionalProperties`. The equivalent but canonical schema on the right expresses all property names as regular expressions, simplifying the subschema check. Note that the regular expression `'([^a]+a|a.).*'` describes keys that contain an `'a'`, but that are not `'a'` itself. This handles the requirement that patternProperties are relevant only for keys not provided as properties.

Canonicalizing enumerations. In full JSON Schema, enumerations of values may be heterogeneous, i.e., contain multiple different types. Our canonicalization ensures that enumerations are homogeneous by transforming any heterogeneous enumeration into a disjunction of multiple homogeneous enumerations.

3.2 Simplifying Combined Schemas

The canonicalization described so far ensures that schemas consist of simpler schemas that each describe only one type and that some keywords of the JSON Schema language are used in specific ways only. The following describes the second step of our approach: a simplifier that further reduces the complexity of JSON schemas. Column “Simplified” of Table 1 summarizes the properties that the simplifier establishes. The simplifier eliminates many cases of enumerations and logical connectives, making the subschema checking rules less complicated. Similar to the canonicalization, the simplifier consists of a set of transformation rules that transform schemas into semantically equivalent yet simpler schemas.

Eliminating non-Boolean enumerations. The first set of transformations eliminates enumerations, replacing them with type specific restriction keywords. For instance, in Figure 2c, the enumerated empty string value is compiled into the regular expression `'^$'` before computing its complement `'.+'` in Figure 2e. Numeric enumerations are converted into the disjunction of appropriately bounded ranges. For example, `{enum:[1,2,4]}` becomes `{anyOf:[{type:integer, minimum:1,maximum:2},{type:integer,minimum:4,maximum:4}]}`. Using disjunction here is appropriate as the schemas are non-overlapping.

For structured types, i.e., arrays and objects, the approach pushes down enumerations to components, until it eventually reaches primitive types, where the enumerations get eliminated. The simplifier does not eliminate Boolean enumerations as the space of values is finite and there is no other way to specify the true and false values.

Simplifying logical connectives. The second set of transformations simplifies logical connectives by eliminating the `not`, `allOf`, and `anyOf` keywords in all but a few specific cases, and by completely eliminating the `oneOf` keyword. The approach eliminates negations by using type-specific ways to express negation and by applying De Morgan's rule. An example is the transformation of Figure 2c to Figure 2e, where the complement of a string schema introduces schemas of all non-string types. We keep negation in numbers, arrays, and objects because JSON Schema is not closed under complement for these schema types. To eliminate `allOf`, the approach uses type-specific ways to reason about intersections of types. For example, eliminating the `allOf` in Figure 2e yields the simplified schema in Figure 2d. We choose not to push intersections through negations because we prefer the end result of simplification to resemble disjunctive normal form to the extent possible. We reduce the cases in which `anyOf` occurs using type-specific ways to reason about unions of types. For example, the following schema that specifies strings using regular expressions gets simplified by computing the union of the two regular expressions:

```
{'anyOf': [{'type': 'string',
            'pattern': '.+'},                 {'type': 'string',
           {'type': 'string',       →          'pattern': '.+'}
            'pattern': 'a'}]}
```

The transformations keep some `anyOf`s since JSON Schema numbers, arrays, and objects are not closed under union. E.g., the union of `{'type':'number','minimum':0}` and `{'type':'number','multipleOf':1}` is $\mathbb{R}^+ \cup \mathbb{Z}$, which JSON schema cannot express without `anyOf`.

Finally, we eliminate `oneOf` by rewriting it as a disjunction of (partially negated) conjunctions, and then simplifying the result as above. For example, assuming three subschemas $s1, s2, s3$:

```
                  {'anyOf':[
{'oneOf':           {'allOf': [s1, {'not': s2}, {'not': s3}]},
  [s1, s2, s3]}  →  {'allOf': [{'not': s1}, s2, {'not': s3}]},
                    {'allOf': [{'not': s1}, {'not': s2}, s3]}]}
```

3.3 JSON Subschema Checking

Given two canonicalized and simplified schemas, the final step is to check whether one schema is a subtype of the other. The basic idea is to extract pairs of type-homogeneous schema fragments from the schemas, and then check the subtype relation for each pair. This approach is enabled by canonicalization and simplification, which allows for a compact set of rules that define the subtype checking.

Figure 4 shows these inference rules. Each rule describes the preconditions (above the bar) for concluding that there is a subtype relation (below the bar) of two schemas, where <: means "is subtype of". Given two schemas s and t, our implementation recursively applies these rules to fragments of s and t. The approach concludes that s <: t if and only if there is a subtype relation between all pairs of extracted fragments. The notation $s.k$ indicates access of keyword k in schema s. For any JSON schema s, helper function $dom(s)$ returns its property names, i.e., the set of keys in the key-value map s. The notation `[...]` indicates a JSON array and the notation `{...}` indicates a JSON object. The notation $a \parallel b$ is a default operator that returns a if it is defined and b otherwise.

Rule *null* simply says schemas that allow only the null value are subtypes of each other. Rules *boolean* and *string* state that a schema s_1 that allows a subset or the same set of values as a schema s_2 is a subschema of s_2. Rule *number* handles complexity due to `multipleOf` constraints. The simplifier cannot push negation through `multipleOf` constraints, and it cannot combine `allOf` combinations of such negated schemas. Instead, the rule handles several such constraints on both sides of the relation, with or without negation. We treat simple number schemas as single-element `allOf`s for consistency. This rule verifies that any number allowed by the set of constraints on the left is also allowed by the set of constraints on the right using an auxiliary *subNumber* relation, which is sketched in the following.

The *subNumber* relation first normalizes all schema range bounds by rounding them to the nearest included number that satisfies its `multipleOf` constraint. For each side, it then finds the least and greatest finite bound used. Every unbounded schema is split into two (or three for totally unbounded) schemas: one (or two) that are unbounded on one side, with the least/greatest bound as the other bound. The "middle" part is bounded. All these derived schemas keep the original `multipleOf`. The bounded schemas are all checked (exhaustively if needed). We can separately check the non-negated and negated unbounded schemas, as they do not problematically interact over unbounded sets. If PL and PR are the left and right non-negated schemas, and NL and NR are the left and right negated schemas, verify that the constraints divide each other:

$$\forall_{pl \in PL}, \exists_{pr \in PR},\ pl.\texttt{multipleOf} \bmod pr.\texttt{multipleOf} = 0$$
$$\forall_{nr \in NR}, \exists_{nl \in NL},\ nr.\texttt{multipleOf} \bmod nl.\texttt{multipleOf} = 0$$

For example, because 3 divides 9 and 2 divides 4, we have:

```
{'allOf':[                                  {'allOf':[
 {'type': 'number',                          {'type': 'number',
  'multipleOf': 9},                    <:     'multipleOf': 3},
 {'type': 'number',                          {'type': 'number',
  'not': {'multipleOf': 2}}]}                 'not': {'multipleOf': 4}}]}
```

This somewhat complex scheme is necessary to handle inclusions on finite ranges that cannot be handled by simple factoring. For example, `{multipleOf:3}<:{allOf:[{multipleOf:3},{multipleOf:2}]}` holds on the bounded range `[4,8]`. Splitting unbounded ranges into parts enables correctly comparing the entirely unbounded and bounded parts. Similarly, if only `{multipleOf:3}` is bounded to range `[4,8]` while `{allOf:[{multipleOf:3},{multipleOf:2}]}` is unbounded, splitting enables us to conclude (correctly) that the subschema relation holds.

Rule *array* checks two array schemas. The size bounds of the left array should be within the size bounds of the right array. Additionally, the schema of every item specified in the former needs to be a subschema of the corresponding specification in the latter. If a schema is not explicitly provided, the schema provided by `additionalItems` is used. Recall that canonicalization adds in a default `additionalItems` schema if it was not specified. Additionally, if the right side specifies that the items must be unique, then the left needs to either specify the same or implicitly enforce this. For example, `{'type':'array','items':[{'enum':[0]},{'enum':[1]}]}` is a subschema of `{'type':'array','uniqueItems':true]}`. The *allDisjointItems* predicate checks for this by first obtaining the set of all the effective item schemas: every item schema for an index within the specified min/max bounds, and `additionalItems` if any allowed indices

$$\text{null}\ \frac{s_1.\texttt{type} = \texttt{null} \qquad s_2.\texttt{type} = \texttt{null}}{s_1 <: s_2} \qquad\qquad \text{boolean}\ \frac{s_1.\texttt{type} = \texttt{boolean} \qquad s_2.\texttt{type} = \texttt{boolean} \qquad s_1.\texttt{enum} \subseteq s_2.\texttt{enum}}{s_1 <: s_2}$$

$$\text{string}\ \frac{s_1.\texttt{type} = \texttt{string} \qquad s_2.\texttt{type} = \texttt{string} \qquad s_1.\texttt{pattern} \subseteq s_2.\texttt{pattern}}{s_1 <: s_2}$$

$$\text{number}\ \frac{\begin{array}{c}\forall i \in \{1,\ldots,k\}, \texttt{not} \notin dom(s_i) \wedge s_i.\texttt{type} = \texttt{number}\\ \forall i \in \{k+1,\ldots,n\}, \texttt{not} \in dom(s_i) \wedge s_i.\texttt{not.type} = \texttt{number}\\ \forall i \in \{1,\ldots,l\}, \texttt{not} \notin dom(t_i) \wedge t_i.\texttt{type} = \texttt{number}\\ \forall i \in \{l+1,\ldots,m\}, \texttt{not} \in dom(t_i) \wedge t_i.\texttt{not.type} = \texttt{number}\\ subNumber(([s_1,\ldots,s_k],[s_{k+1},\ldots,s_n]),([t_1,\ldots,t_l],[t_{l+1},\ldots,t_m]))\end{array}}{\{\texttt{allOf}\texttt{:}[s_1,\ldots,s_k,s_{k+1},\ldots,s_n]\} <: \{\texttt{allOf}\texttt{:}[t_1,\ldots,t_l,t_{l+1},\ldots,t_m]\}}$$

$$\text{array}\ \frac{\begin{array}{c}\begin{array}{cc} s_1.\texttt{type} = \texttt{array} & s_2.\texttt{type} = \texttt{array}\\ s_1.\texttt{minItems} \geq s_2.\texttt{minItems} & s_1.\texttt{maxItems} \leq s_2.\texttt{maxItems}\\ s_1.\texttt{items} = [s_{1_1},\ldots,s_{1_k}] & s_2.\texttt{items} = [s_{2_1},\ldots,s_{2_m}]\end{array}\\ \forall i \in \{0,\ldots,max(k,m)+1\},\ (s_{1_i} \parallel s_1.\texttt{additionalItems}) <: (s_{2_i} \parallel s_2.\texttt{additionalItems})\\ s_2.\texttt{uniqueItems} \implies (s_1.\texttt{uniqueItems} \vee allDisjointItems(s_1))\end{array}}{s_1 <: s_2}$$

$$\text{object}\ \frac{\begin{array}{c}\begin{array}{cc} s_1.\texttt{type} = \texttt{object} & s_2.\texttt{type} = \texttt{object}\\ s_1.\texttt{minProperties} \geq s_2.\texttt{minProperties} & s_1.\texttt{maxProperties} \leq s_2.\texttt{maxProperties}\end{array}\\ s_1.\texttt{required} \supseteq s_2.\texttt{required}\\ \forall p_1\texttt{:}s_{p_1} \in s_1.\texttt{patternProperties}, p_2\texttt{:}s_{p_2} \in s_2.\texttt{patternProperties}, p_1 \cap p_2 \neq \emptyset \implies s_{p_1} <: s_{p_2}\end{array}}{s_1 <: s_2}$$

$$\text{non-overlapping anyOf}\ \frac{\forall i \in \{1,\ldots,n\}, \exists j \in \{1,\ldots,m\}, s_i <: t_j \qquad nonOverlapping([t_1,\ldots,t_m])}{\{\texttt{anyOf}\texttt{:}[s_1,\ldots,s_n]\} <: \{\texttt{anyOf}\texttt{:}[t_1,\ldots,t_m]\}} \qquad\qquad \text{uninhabited}\ \frac{\neg\, inhabited(s_1)}{s_1 <: s_2}$$

Figure 4: JSON Schema subtype inference rules.

are unspecified. It then verifies that the conjunction of all pairs of effective item schemas are uninhabited.

Rule *object* checks two object schemas. It first verifies that the number of properties of both sides have the appropriate relation and that the left side requires all the keys that the right side requires. Next, for every regular expression pattern p_1 on the left, if there is an overlapping regular expression pattern p_2 on the right, it checks that the corresponding schemas are subschemas. This check can be done one pattern at a time since canonicalization eliminates overlapping pattern properties. We note that regular expressions are well known to be closed under union, intersection, and complement [38], and deciding inclusion for regular expressions is decidable[1].

Rule *non-overlapping anyOf* handles anyOf schemas for the cases where simplification eliminates overlapping unions. Helper function *nonOverlapping* checks for unions of arrays and objects and conservatively assumes that those might overlap. In the non-overlapping case, it suffices to check the component schemas independently. For each schema on the left, we require a same-typed superschema on the right.

Finally, rule *uninhabited* states that an uninhabited schema is a subtype of any other schema. "Uninhabited" here means that no legal JSON data exists for the schema. This rule is the only rule that checks schemas of possibly different types, since there is no valid data that would constrain the type. We define simple rules for uninhabitedness, e.g., checking for numerical schemas that the minimum is larger than the maximum, which are elided from this exposition. Like the other parts of this paper, the uninhabitedness checker is intended (but not proven) to be sound, but not complete. Baazizi et al.'s witness generation algorithm could be used for a more complete inhabitedness check [20].

Unsupported Features. Our approach does not handle the following corner cases of the JSON Schema specification: negation, disjunction, and enumeration of object and array schemas; non-regular regex patterns; and recursive references. Our approach detects these corner cases and returns "unknown" for them. Section 5.3.3 shows that these cases occur in only 6.5% of the real-world schema pairs we tested with. Addressing them is future work.

4 IMPLEMENTATION

We implemented our subschema checker as an open-source Python library (https://github.com/ibm/jsonsubschema). The implementation builds upon the *jsonschema library* [8] to validate schemas before running our subtype checking, the *greenery library* [9] for computing intersections of regular expressions, and the *jsonref library* [11] for resolving JSON schema references.

5 EVALUATION

This section evaluates our JSON subschema checker, which we refer to as *jsonsubschema*. It addresses the following research questions:

RQ$_1$ How effective is *jsonsubschema* in finding real bugs?
RQ$_2$ How correct and complete is *jsonsubschema*?
RQ$_3$ How does our approach compare against prior work?
RQ$_4$ How efficient is *jsonsubschema*?

5.1 Experimental Setup

We evaluate our approach on four datasets of JSON schemas listed in Table 2. The datasets cover different application domains and different ways of using JSON schemas. Snowplow is a service for

[1] Asking if r_1 is included in r_2 is the same as asking if the complement of r_2 intersected with r_1 is equivalent to the regular expression that accepts nothing. This can be checked by converting to a finite automate, minimizing, and checking for equivalence [37].

Table 2: Datasets of JSON schemas used for evaluation.

Project	Description	Schemas (incl. versions)
Snowplow	Data collection and live-streaming of events [55]	306 (382)
Lale	Automated machine learning library [21]	1,444 (*NA*)
Washington Post (WP)	Content creation and management [13]	165 (2,728)
Kubernetes (K8)	Containerized applications [16]	1,336 (104,453)
Total		3,251 (107,563)

data collection and aggregation for live-streaming of event data [2]. The service aggregates data from various sources and uses JSON schemas to specify data models, configurations of several components, etc. [55]. Lale is a Python library for type-driven automated machine learning [21]. It uses JSON schemas to describe the inputs and outputs of machine-learning (ML) operators, as well as standard ML datasets. The Washington Post dataset is a collection of schemas describing content used by the Washington Post within the *Arc Publishing* content creation and management system [13]. The final dataset comprises JSON schemas extracted from the OpenAPI specifications for Kubernetes [16], a system for automating deployment, scaling, and management of containerized applications [4]. The last column of Table 2 shows how many schemas each of these projects has, and how many schemas there are when considering each version of a schema as a separate schema. The total number of schemas is 3,251, and 107,563 including versions.

Many of the schemas are of non-trivial size, with an average of 56KB and a maximum of 1,047KB. To validate the correctness of the canonicalization and simplification steps of *jsonsubschema*, we also use the official test suite for JSON Schema draft-04 [10]. It contains 146 schemas and 531 JSON documents that fully cover the JSON Schema language. All experiments used Ubuntu 18.04 (64-bit) on an Intel Core i7-4600U (2.10GHz) machine with 12GB RAM.

5.2 Effectiveness in Detecting Bugs (RQ$_1$)

To evaluate the usefulness of *jsonsubschema* for detecting bugs, we consider two real-world scenarios where we could clearly determine the expected relationship between pairs of schemas using an independent source of ground truth. In both examples, the correctness of software requires a specific subschema relation to hold.

5.2.1 Schema Evolution Bugs in Snowplow. The first bug detection scenario checks whether evolving schemas in the Snowplow project may break clients that rely on the backward compatibility of schemas. Snowplow maintains versioned schemas that specify data sources and configurations [55]. To ensure backward compatibility of clients and to avoid unnecessary updates of client code, the project adopts semantic version numbers of the form *major.minor.patch* [6]. In a blog post, the developers of Snowplow express the goal of using these schemas and versions to make interactions less error-prone [3]—in other words, to avoid data compatibility bugs. For each schema evolution, we check whether the schema change is consistent with the version number change. Specifically, a backward compatible schema fix corresponds to a patch increment, a change that adds functionality in a backward-compatible way to a minor increment, and a change that breaks backward compatibility to a major increment.

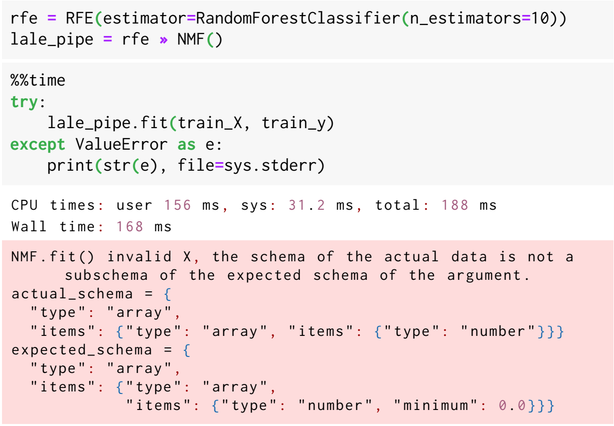
```
{'properties': {
   'event':
     {'type': 'object'},
   'error':
     {'type': 'string'},
   ...}
 'required':
   ['event', 'error'],
 'additionalProperties':
   false }
```

Version 1.0.0

```
{'properties': {
  'payload':
    {'type': 'object'},
  'failure':
    {'type': 'string'},
  ...}
 'required':
   ['payload', 'failure'],
 'additionalProperties':
   false }
```

Version 1.0.1

Figure 5: *Snow-1:* A schema evolution bug in Snowplow.



```
rfe = RFE(estimator=RandomForestClassifier(n_estimators=10))
lale_pipe = rfe » NMF()
```

```
%%time
try:
    lale_pipe.fit(train_X, train_y)
except ValueError as e:
    print(str(e), file=sys.stderr)
```

```
CPU times: user 156 ms, sys: 31.2 ms, total: 188 ms
Wall time: 168 ms
```

```
NMF.fit() invalid X, the schema of the actual data is not a
     subschema of the expected schema of the argument.
actual_schema = {
  "type": "array",
  "items": {"type": "array", "items": {"type": "number"}}}
expected_schema = {
  "type": "array",
  "items": {"type": "array",
           "items": {"type": "number", "minimum": 0.0}}}
```

Figure 6: Dataset error check example in Lale.

For this experiment, we consider all schemas in the Snowplow project that have at least two versions, and then apply *jsonsubschema* to all pairs of consecutive schema versions. For each pair of two consecutive versions, we then check if the subtype relation found by *jsonsubschema* is consistent with the version numbering.

Our approach detects five schema evolution bugs, summarized in the top part of Table 3. We summarize multiple similar bugs into a single one for space reasons. For example, in *Snow-1* (Figure 5), two object properties changed their names. This change breaks backward compatibility for old data, hence, the major version number should have been incremented. The developers of Snowplow confirmed all reported bugs in Table 3 and acknowledged that specifically *Snow-1* and *Snow-2* are severe. One developer wrote that "Our long-term plan is to implement an automatic algorithm to recognize versioning and clarify/formalise specification for known corner-cases". Our approach provides this kind of tool, and could help avoid schema evolution bugs in the future.

5.2.2 Incorrect ML Pipelines in Lale. As a second real-world usage scenario, we apply *jsonsubschema* to type-check machine learning pipelines implemented in Lale [21]. Lale uses JSON schemas to describe both ML operators and ML datasets. An ML pipeline is a graph where nodes are operators and edges are dataflow. Lale uses *jsonsubschema* to check whether the root operators of an ML pipeline are compatible with an input dataset, as well as whether on all edges the intermediate dataset from the predecessor is compatible with the successor.

Table 3: 43 real-world bugs detected by *jsonsubschema*. Issue ids and commits removed for double-blind review.

Bug Id	Description	Bug Report Status
	Bugs in Snowplow schema versions	
Snow-1	*Breaking change.* Wrong increment of Patch version; should increment Major version instead.	Confirmed
Snow-2	*Wrong version.* Wrong update of Minor and Patch versions; should increment Patch instead.	Confirmed
Snow-3	*Spurious version increment.* Increment of Major version; should increment Patch instead.	Confirmed
Snow-4–5	*Spurious version increment.* Increment of Minor version; should increment Patch instead.	Confirmed
	Bugs in schemas of machine learning operators in Lale	
Lale-1–2	Classifiers output either string or numeric labels; output schemas should be union of arrays not array of unions.	Fixed
Lale-3–14	Classifiers should allow output labels to be strings or numeric instead of numeric only.	Fixed
Lale-15–32	Classifiers should allow output labels to be Booleans, beside strings or numeric.	Fixed
Lale-33	Using the empty schema in a union is too permissive and implies the empty schema, which validates everything.	Fixed
Lale-34–38	Using the empty schema as an output schema causes problems when used as input to the next operator.	Fixed

```
{'type': 'array',
 'items': {
  'anyOf':[
   {'type': 'number'},
   {'type': 'string'}]}}
```

Wrong schema

```
{'anyOf': [
  {'type': 'array',
   'items': {'type': 'number'}},
  {'type': 'array',
   'items': {'type': 'string'}}]}
```

Correct schema

Figure 7: *Lale-1:* Wrong schema for an ML operator in Lale.

Figure 6 shows an example. Lines 1–2 configure an ML pipeline with two operators, RFE (recursive feature elimination) and NMF (non-negative matrix factorization). The pipe combinator (») introduces a dataflow edge between these two operators. Line 5 tries to train the pipeline on a given dataset. Internally, Lale uses *jsonsubschema* to check whether the output schema of RFE is a subschema of the input schema of NMF. In this example, that check quickly yields an error message, since RFE returns an array of arrays of numbers, but NMF expects an array of arrays of *non-negative* numbers. In this example, the subschema check saved time compared to the approach of first training RFE, which can take minutes, and only then triggering an exception in NMF due to negative numbers.

We ran Lale's regression test suite with instrumentation to log all subschema checks and counted 2,818 unique schema pairs being checked. In two years of production use of *jsonsubschema*, the checks have revealed many bugs, 38 of which are summarized in the lower part of Table 3. All bugs have been fixed in response to finding them with *jsonsubschema*. For example, in *Lale-1* (Figure 7), a subschema check reveals that the output of a classifier could either be numeric or string labels, but no mixing of the two kinds.

The approach detects 43 bugs, most of which are already fixed.

5.3 Correctness and Completeness (RQ$_2$)

We aim for soundness, i.e., giving a correct answer whenever not returning "unknown", while being as complete as possible, i.e., trying to cover as many JSON Schema features as possible. The following evaluates to what extent our approach and implementation achieves these goals.

5.3.1 Canonicalization and Simplification. Canonicalization and simplification transform a given schema into a simpler yet semantically equivalent schema. To check this property, we use the official JSON Schema test suite to validate JSON documents against schemas before and after the two transformation steps. Specifically, we check whether:

$$\forall s, \forall d \; valid(d, s) \Leftrightarrow valid\big(d, simple\big(canonical(s)\big)\big)$$

for schema s and its associated JSON document d in the JSON Schema test suite. In all cases where *jsonsubschema* yields a canonicalized schema, this new schema passes all relevant tests in the JSON Schema test suite, with one exception. This single case is due to an ambiguity in the specification of JSON Schema about the semantics of the allOf connector combined with the additionalProperties object constraint.

5.3.2 Ground Truth. To evaluate the correctness and completeness of the approach as a whole, we establish a ground truth by sampling and manually inspecting 300 schema pairs. We sample schema pairs from all schemas listed in Table 2 using two strategies. The first strategy focuses on schemas likely to be related to each other. For datasets where the schemas have versions (Snowplow, WP, K8), we consider all pairs of schemas that have the same name but a different version. For the Lale dataset, we consider schema pairs where one schema describes an ML dataset and the other schema describes an ML operator that could, in principle, be applied to the dataset. For each of the four projects, we randomly sample 50 schema pairs. The second strategy focuses on supposedly unrelated schemas. To this end, we consider the union of schemas from all four projects and randomly sample 100 pairs. For both strategies, we exclude schemas that are syntactically invalid and schemas that contain an invalid or unresolvable URI in a $ref, and our tool inlines all non-recursive URIs.

The sampling results in 300 schema pairs (Table 4). For each sampled pair, we carefully inspect both schemas to decide whether they are in a subtype relationship. Out of the 300 schema pairs, 108 are in subtype relation (<:) while 192 are not ($\not<:$). Overall, the sampling and manual inspection took around 40 working hours. We envision that the resulting ground truth serves not only for this paper, but also as a benchmark for future work.

5.3.3 Precision, Recall, Correctness. Based on the manually established ground truth, we measure the precision, recall, and correctness of our approach. Because the approach has three possible outputs for a given schema pair—subschema, not subschema, and unknown—each pair belongs into exactly one of the following six categories: true positive (TP) if the output is subschema and matches the ground truth; true negative (TN) if the output is not

Table 4: Effectiveness of *jsonsubschema* and comparison to the existing *issubset* tool.

	Ground truth			*jsonsubschema*						*issubset*					
Dataset	Pairs	$<:$	$\not<:$	TP	TN	FP	FN	$?_{<:}$	$?_{\not<:}$	TP	TN	FP	FN	$?_{<:}$	$?_{\not<:}$
Snowplow	50	17	33	14	29	0	0	3	4	13	32	1	4	0	0
WP	50	35	15	31	4	0	0	4	11	31	3	4	0	4	8
K8	50	31	19	31	19	0	0	0	0	23	17	2	8	0	0
Lale	50	22	28	22	28	0	0	0	0	16	28	0	6	0	0
Unrelated	100	3	97	3	97	0	0	0	0	3	57	40	0	0	0
Total	300	108	192	101	177	0	0	7	15	86	137	47	18	4	8
Precision				100%						64.7%					
Recall				93.5%						79.6%					
Correctness				100%						77.4%					

subschema and matches the ground truth; false positive (FP) if the output is subschema but the ground truth is not subschema; false negative (FN) if the output is not subschema but the ground is subschema; unknown for subschema ($?_{<:}$) if the approach does not give an answer and the ground truth is subschema; and unknown for not subschema ($?_{\not<:}$) if the approach does not give an answer and the ground truth is not subschema. Precision is defined as the percentage of correctly identified subschema pairs among all pairs the approach identifies as a subschema:

$$precision = \frac{TP}{TP + FP}$$

Recall is the percentage of correctly identified subschema pairs among all pairs that indeed are subschemas:

$$recall = \frac{TP}{TP + FN + ?_{<:}}$$

As an additional metric, we compute correctness as the percentage of correct answers among all answers the approach gives:

$$correctness = \frac{TP + TN}{TP + TN + FP + FN}$$

The middle part of Table 4 shows the results of our *jsonsubschema* tool w.r.t. the ground truth. Across the 101 cases where *jsonsubschema* reports two schemas to have a subschema relation, all are correct. That is, the approach has 100% precision. Across all 101+177=278 cases where the approach gives an answer, again all answers are correct. That is, the correctness of the approach is 100%. Both results match our design goal of ensuring the correctness of the algorithm.

As mentioned in Section 3, there are some cases that *jsonsubschema* does not decide. In practice, we found that such cases are rather rare, with a recall of 93.5%. That is, the approach gets close to completeness, i.e., supporting all real-world subschema pairs, but there remain some JSON schemas where the approach does not give an answer. There are two reasons for not reaching full recall. For 14 out of the 22 pairs, at least one schema uses a recursive set of references through the `$ref` keyword. For the remaining eight pairs, *jsonsubschema* could not decide on the sub-schema relation due to a negated object. Both reasons are limitations of our approach, which we consider to be non-trivial to address.

5.3.4 Pervasiveness of Validation Keywords and Supported Features. Figure 8 shows the frequency of validation keywords across all schemas in the four datasets in Table 2. Validation keywords on the x-axis are sorted by their relevance to each schema type and according to the order of keywords in Table 1. The figure shows that *jsonsubschema* indeed supports the majority of JSON schema features used in practice.

We observe that JSON schema types `null` and `string` are the two most prevalent schema types present in the dataset. Both types are fully supported in the subtype checking performed by *jsonsubschema* as indicated by the color code in Figure 8. The keywords `properties` and `required` for specifying constrains on a JSON `object` show up next on the order of the number of use cases. *jsonsubschema* fully supports `properties` while the `required` keyword is supported whenever it is not used in union schemas or negated schemas. In general, disjunction of schemas happens rarely (366 occurrence among millions of occurrences of other keywords), while negated schemas are not used at all in our dataset.

The counts in Figure 8 are for schemas that do not use the negated schema keyword `not` at all, which is also evident from its frequency being 0. The reason is that in the dataset of schemas, there is no single use of a negated schema. In fact, the use of negation in JSON schemas is indeed highly discouraged since the purpose of schema validation is to constrain what is allowed rather than filtering out what is disallowed. The NSA security guidelines for using JSON schemas also advises against using negated schemas for the same reason.[2] That said, *jsonsubschema* still supports the negation of all basic types (null, boolean, string, integer, and number) except for the union and negation of numeric schemas with a `multipleOf` constraint. Overall, this shows that the incompleteness of *jsonsubschema* rarely affects its usability on a large dataset of real-world schemas.

The only feature that is not supported at the moment is recursive references in schemas using `$ref`. Although our approach currently does not handle recursive schemas, we know theoretically that subtyping recursive types is decidable [17]. The *jsonsubschema* tool reports an error and terminates without yielding a decision when an unsupported JSON schema feature is encountered.

> The approach is correct whenever it gives an answer (100% precision and 100% correctness) and can handle the majority of a sample of real-world schema pairs (93.5% recall). Additionally, *jsonsubschema* supports the majority of those JSON Schema features that are widely used in practice.

5.4 Comparison to Existing Work (RQ$_3$)

To our knowledge, this is the first academic work to address the problem of JSON subschema checking, and the first work at all to describe how to address this problem for a large subset of JSON Schema. The closest developer tool we could find is *issubset* [35], a tool that states the same goal as ours: "Given a schema defining the output of some process A, and a second schema defining the input of some process B, will the output from A be valid input for process B?"

We compare against the most recent version (1.1.25) of *issubset* by running the tool on all schema pairs in our ground truth (Section 5.3.2). Initially, the tool failed on almost half the pairs in

[2] https://apps.nsa.gov/iaarchive/library/reports/security_guidance_for_json.cfm

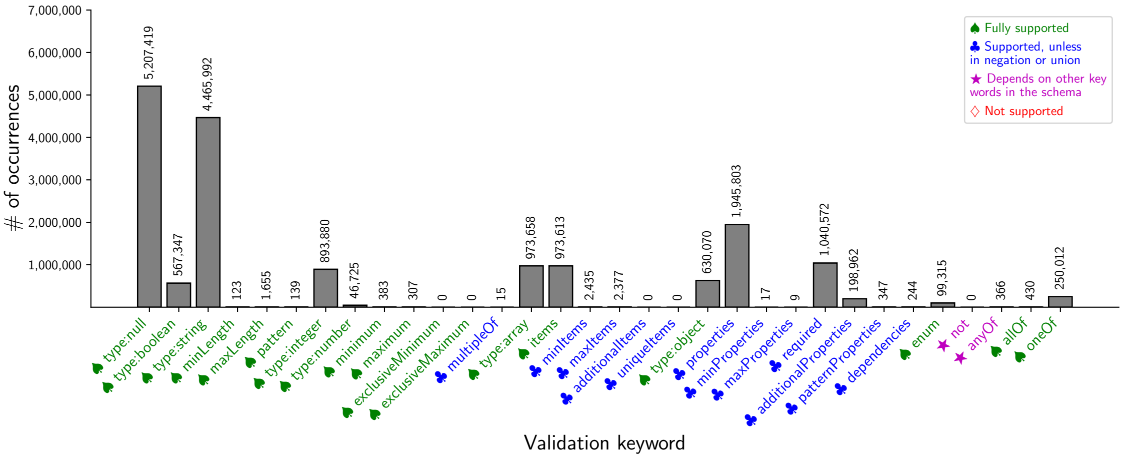


Figure 8: Prevalence of JSON schema validation keywords in practice and supported features in *jsonsubschema*.

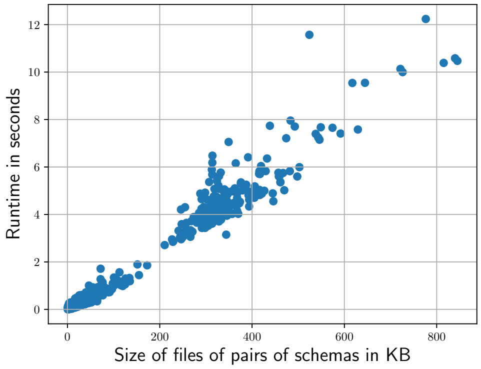


Figure 9: Efficiency of subschema checking.

our dataset with an error complaining about the schemas using the JSON Schema draft04, while the tool supports schemas starting from draft05, although the *issubset* documentation does not describe such a limitation. After careful inspection of the pairs of schemas in our ground truth, we find that none of the schemas uses any JSON Schema feature that changed in the later versions of the language. We hence set the JSON Schema version number in those schemas to draft05, enabling the *issubset* tool to check them.

The right part of Table 4 shows the results. *issubset* produces 86 true positives, which means it indeed captures some of the semantics of the subtyping relation. However, *issubset* also produces 46 false positives and 18 false negatives, i.e., gives a wrong answer to a subtype query. Overall, the existing tool achieves a precision of 64.7% and a recall of 79.6%, which is significantly lower than our approach. Additionally, its overall correctness is at 77.4%, i.e., clearly below ours.

To better understand *issubset*'s limitations, we inspect their code and test it on simple schemas. Although the tool performs some semantic checks, e.g., it correctly reports `{'type':'integer'}` <: `{'type':'number'}`, it lacks the ability to capture the richness of JSON Schema in many ways. For instance, it is oblivious to uninhabited schemas, such as `{'type':'string','enum':[1]}`, and it fails to detect `{'type':['string','null']}` <: `{'type':['null','string']}`.

> Our approach outperforms the best existing subschema checker in terms of precision (100% vs. 64.7%), recall (93.5% vs. 79.6%), and overall correctness (100% vs. 77.4%).

5.5 Efficiency (RQ$_4$)

To evaluate how fast our subschema checker is in practice, we measure the time taken by subschema checks on a random sample of 1,000 pairs of non-equal schemas from Table 2. We took every time measurement ten times and report the average. Figure 9 shows the size of pairs of schema files in KB against the time subschema checking takes in seconds. In most cases, our subschema checker terminates within a few seconds for moderately sized schemas. Our subschema approach is lazy and terminates on the first violation of a subtyping rule. That said, and since we envision subschema checks to have wide range of applications, we will explore how to improve on this performance, for instance, by on-demand canonicalization.

> A subschema check takes up to several seconds, and the time scales linearly with the schema size.

6 THREATS TO VALIDITY

The datasets in Table 2 may not be representative of all JSON schema pairs for which users may want to check compatibility. The variety and size of the datasets and the fact that the developers confirmed or fixed all reported bugs reduces this threat, but we do not claim that our results necessarily generalize to all other JSON schemas.

The results in Table 4 use manually-created ground truth. Hence, there is a possibility of human error. This is mitigated by also running a variety of other tests that can be validated automatically without manual human effort.

7 RELATED WORK

7.1 JSON Schema and Schema Subtyping

Practitioners have significant interest in reasoning about the subtype relation of JSON schemas. Section 5.4 has an experimental comparison against the strongest competitor among the available tools, *issubset* [35], which was developed concurrently with our work [33]. Another closely related tool [15] relies on simple syntactic checks. For example, that work considers a change as a breaking change whenever a node is removed from the schema. As illustrated by Figure 2, removing nodes (or replacing them by others) may yield not only subtypes but even equivalent schemas. Yet another existing tool [14] checks whether two schemas are equivalent but does not address the subtyping problem. Baazizi et al. [20] describe an algorithm for JSON Schema witness generation, which could answer $s <: t$ by showing that `{allOf:{not:`s`}, `t`}` has no witness; however, the tool from the paper is not available.

Pezoa et al. [46] formally define the syntax and semantics of JSON Schema, including the JSON validation problem. An alternative formulation of JSON validation uses a logical formalism [25]. Baazizi et al. [19] address the problem of inferring schemas for irregular JSON data. None of the above pieces of work addresses the subschema problem. There are other schema definition languages for JSON besides JSON Schema, e.g., Avro [1]. Protobuf [5] is Google's data exchange format, which borrows several features from JSON schema. Our work might help define subtype relations for these alternative languages.

The Swagger (OpenAPI) specification [7] uses JSON Schema to define the structure of RESTful APIs. The *swagger-diff* tool [12] aims at finding breaking API changes through a set of syntactic checks, but does not provide the detailed checks that we do. Our *jsonsubschema* tool could be integrated as part of the pipeline to check for backward compatibility.

7.2 Type Systems for XML, JavaScript, Python

Semantic subtyping [27] handles Boolean connectives on types by using a disjunctive normal form similar to that in this paper. It was developed in the context of CDuce, a functional language for working with XML [23]. Subschema checking for XML, called schema containment, is also addressed in [56]. XDuce is a static language for processing XML documents using XML schemas as types [39] which makes use of "regular expression types" [40]. Our work differs in working on JSON, not XML, which has a different feature set as described in Table 1. Also these approaches treat XML as tree automata, shown to be less expressive than JSON Schema [46].

Both JavaScript and Python have a convenient built-in syntax for JSON documents. Furthermore, there are type systems retrofitted onto both JavaScript [24] and Python [49]. Therefore, a reasonable question to ask is whether JSON schema subtype queries could be decided by expressing JSON schemas in those languages and then using the subtype checker of those type systems. Unfortunately, this is not the case, since JSON Schema contains several features that those type systems cannot express, such as negation, `multipleOf` on numbers, and `pattern` on strings.

7.3 Applications of Subschema Checks

One application of JSON subschema is statically reasoning about breaking changes of web APIs. A study of the evolution of such APIs shows that breaking changes are frequent [42]. Another study reports that breaking changes of web APIs cause distress among developers [29]. Since JSON schemas and related specifications are widely used to specify web APIs, our approach can identify breaking changes statically instead of relying on testing (Section 5.2.1).

Data validation for industry-deployed machine learning pipelines is crucial as such pipelines are usually retrained regularly with new data. To validate incoming data, Google TFX [22] synthesizes a custom data schema based on statistics from available data and uses this schema to validate future data instances [26]. Amazon production ML pipelines [52] offer a declarative API to define desired properties of data, which are then checked on real-time data. Both systems are missing an explicit notion of schema subtyping. For instance, TFX uses versioned schemas to track the evolution of inferred data schemas, and reports back to the user whether to update to a more (or less) permissive schema based on the historical and new data instances [22]. Lale uses JSON schemas to specify both correct ML pipelines and their search space of hyperparameters [21]. The ML Bazaar also specifies ML primitives via JSON [54]. Another type-based system for building ML pipelines is described by [47]. These systems can benefit from JSON subschema checking to avoid running and deploying incompatible ML pipelines (Section 5.2.2).

7.4 General-Purpose Bug Detection

Static [18, 30, 32, 41], dynamic [36, 44, 48], and hybrid [34, 53] program analysis techniques are widely used. These approaches are designed to find bugs in program source code and observed runtime behavior. Also, statically detecting defects in cloud server systems [28] and dynamically in data-intensive applications [57] are related lines of work. Our work is orthogonal to the above as it focuses on finding data compatibility bugs at the data specification level. This class of bugs is programming-language independent and therefore *jsonsubschema* complements standard bug finding techniques.

8 CONCLUSION

This paper addresses a class of data compatibility bugs in applications that describe their data with JSON schemas. The core of the approach is a novel subtype checker for such schemas. It addresses the various language features of JSON Schema by first canonicalizing and simplifying schemas, and by then type checking pairs of schema fragments that each describe data of a single type. The subtype checker answers the subtype question correctly whenever it gives an answer, achieving 100% precision and 93.5% recall for schemas that occur in the wild, clearly outperforming the best existing work. Applying the approach to detect data compatibility bugs in popular web APIs and ML pipelines reveals 43 previously unknown bugs, most of which have already been fixed. We envision our work to contribute to more reliable software in data-intensive applications across different domains.

ACKNOWLEDGMENTS

This work has been partially supported by the European Research Council (ERC, grant agreements 851895 and 949014), and by the German Research Foundation within the ConcSys and Perf4JS projects.

REFERENCES

[1] [n.d.]. *Apache Avro.* http://avro.apache.org/

[2] [n.d.]. *The enterprise-grade event data collection platform.* https://snowplowanalytics.com/

[3] [n.d.]. *Introducing SchemaVer for semantic versioning of schemas.* https://snowplowanalytics.com/blog/2014/05/13/introducing-schemaver-for-semantic-versioning-of-schemas/

[4] [n.d.]. *Production-Grade Container Orchestration.* https://kubernetes.io/

[5] [n.d.]. *Protocol Buffers.* https://developers.google.com/protocol-buffers

[6] [n.d.]. *Semantic Versioning 2.0.0.* https://semver.org/

[7] [n.d.]. *Swagger/OpenAPI Specification.* https://swagger.io/

[8] 2011. *jsonschema: Implementation of JSON Schema for Python.* https://github.com/Julian/jsonschema

[9] 2012. *greenery: Tools for parsing and manipulating regular expressions.* https://github.com/qntm/greenery

[10] 2012. *JSON Schema Test Suite.* https://github.com/json-schema-org/JSON-Schema-Test-Suite

[11] 2013. *jsonref: Library for automatic dereferencing of JSON Reference objects.* https://github.com/gazpachoking/jsonref

[12] 2015. *Swagger::Diff.* https://github.com/civisanalytics/swagger-diff

[13] 2015. *The Washington Post ANS specification.* https://github.com/washingtonpost/ans-schema

[14] 2017. *JSON Schema Compare.* https://github.com/mokkabonna/json-schema-compare

[15] 2017. *JSON Schema Diff Validator.* https://bitbucket.org/atlassian/json-schema-diff-validator

[16] 2019. *Kubernetes JSON Schemas.* https://github.com/instrumenta/kubernetes-json-schema

[17] Roberto M. Amadio and Luca Cardelli. 1993. Subtyping Recursive Types. *Transactions on Programming Languages and Systems (TOPLAS)* 15, 4 (Sept. 1993), 575–631. http://doi.acm.org/10.1145/155183.155231

[18] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs fixit. In *International Symposium on Software Testing and Analysis (ISSTA).* 241–252.

[19] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Counting types for massive JSON datasets. In *Symposium on Database Programming Languages (DBPL).* 9:1–9:12. https://doi.org/10.1145/3122831.3122837

[20] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Not Elimination and Witness Generation for JSON Schema. In *Conférence sur la Gestion de Données (BDA).* https://arxiv.org/abs/2104.14828

[21] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, and Avraham Shinnar. 2020. Lale: Consistent Automated Machine Learning. In *KDD Workshop on Automation in Machine Learning (AutoML@KDD).* https://arxiv.org/abs/2007.01977

[22] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Conference on Knowledge Discovery and Data Mining (KDD).* 1387–1395. https://doi.org/10.1145/3097983.3098021

[23] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: an XML-centric general-purpose language. In *International Conference on Functional Programming (ICFP).* 51–63. https://doi.org/10.1145/944705.944711

[24] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference for Object-Oriented Programming (ECOOP).* 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[25] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *Symposium on Principles of Database Systems (PODS).* 123–135. https://doi.org/10.1145/3034786.3056120

[26] Eric Breck, Marty Zinkevich, Neoklis Polyzotis, Steven Whang, and Sudip Roy. 2019. Data Validation for Machine Learning. In *Conference on Systems and Machine Learning (SysML).* https://www.sysml.cc/doc/2019/167.pdf

[27] Giuseppe Castagna and Alain Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Symposium on Principles and Practice of Declarative Programming (PPDP).* 198–199. https://www.irif.fr/~gc/papers/icalp-ppdp05.pdf

[28] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. DScope: Detecting Real-World Data Corruption Hang Bugs in Cloud Server Systems. In *Symposium on Cloud Computing (SoCC).* 313–325.

[29] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. 2015. Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software (JSS)* 100 (2015), 27–43. https://doi.org/10.1016/j.jss.2014.10.014

[30] Facebook. 2015. Infer: A tool to detect bugs in Java and C/C++/Objective-C code before it ships. https://fbinfer.com/

[31] Francis Galiegue and Kris Zyp. 2013. JSON Schema draft 04. http://json-schema.org/draft-04/json-schema-validation.html

[32] Google. 2015. Error Prone: static analysis tool for Java. http://errorprone.info/

[33] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2019. Type Safety with JSON Subschema. https://arxiv.org/abs/1911.12651

[34] Brian Hackett and Shu-yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI).* 239–250.

[35] Petter Haggholm. 2019. *is-json-schema-subset.* https://github.com/haggholm/is-json-schema-subset

[36] Sudheendra Hangal and Monica S. Lam. 2002. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering (ICSE).* 291–301.

[37] John E. Hopcroft. 1971. *An n Log n Algorithm for Minimizing States in a Finite Automaton.* Technical Report. Stanford, CA, USA.

[38] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[39] Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A Statically Typed XML Processing Language. *Transactions on Internet Technology (TOIT)* 3, 2 (May 2003), 117–148. https://doi.org/10.1145/767193.767195

[40] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *Transactions on Programming Languages and Systems (TOPLAS)* 27, 1 (Jan. 2005), 46–90. https://doi.org/10.1145/1053468.1053470

[41] S. C. Johnson. 1978. *Lint, a C Program Checker.* Murray Hill: Bell Telephone Laboratories.

[42] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. 2013. How Does Web Service API Evolution Affect Clients?. In *International Conference on Web Services (ICWS).* 300–307. https://doi.org/10.1109/ICWS.2013.48

[43] Sam Newman. 2015. *Building Microservices: Designing Fine Grained Systems.* O'Reilly.

[44] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering (ICSE).* 75–84.

[45] David L. Parnas, John E. Shore, and David Weiss. 1976. Abstract types defined as classes of variables. In *Conference on Data: Abstraction, Definition and Structure.* 149–154. https://doi.org/10.1145/800237.807133

[46] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. 2016. Foundations of JSON Schema. In *International Conference on World Wide Web (WWW).* 263–273. https://doi.org/10.1145/2872427.2883029

[47] Martin Pilat, Tomas Kren, and Roman Neruda. 2016. Asynchronous Evolution of Data Mining Workflow Schemes by Strongly Typed Genetic Programming. In *International Conference on Tools with Artificial Intelligence (ICTAI).* 577–584. https://doi.org/10.1109/ICTAI.2016.0094

[48] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *International Conference on Software Engineering (ICSE).*

[49] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 Types in the Wild: A Tale of Two Type Systems. In *Dynamic Languages Symposium (DLS).* 57–70. https://doi.org/10.1145/3426422.3426981

[50] Eric Redmond and Jim R. Wilson. 2012. *Seven Databases in Seven Weeks.* The Pragmatic Bookshelf.

[51] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. 2016. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In *International Conference on Web Engineering (ICWE).* 21–39. https://doi.org/10.1007/978-3-319-38791-8_2

[52] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating Large-scale Data Quality Verification. In *Conference on Very Large Data Bases (VLDB).* 1781–1794. https://doi.org/10.14778/3229863.3229867

[53] Yannis Smaragdakis and Christoph Csallner. 2007. Combining Static and Dynamic Reasoning for Bug Detection. In *International Conference on Tests and Proofs (TAP).* 1–16.

[54] Micah J. Smith, Carles Sala, James Max Kanter, and Kalyan Veeramachaneni. 2019. The Machine Learning Bazaar: Harnessing the ML Ecosystem for Effective System Development. https://arxiv.org/abs/1905.08942

[55] Snowplow Analytics. 2014. *Central repository for storing JSON Schemas, Avros and Thrifts.* https://github.com/snowplow/iglu-central

[56] Akihiko Tozawa and Masami Hagiya. 2003. XML Schema Containment Checking based on Semi-implicit Techniques. In *International Conference on Implementation and Application of Automata (CIAA).* 213–225. https://doi.org/10.1007/3-540-45089-0_20

[57] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Symposium on Operating Systems Design and Implementation (OSDI).* 249–265.

[58] Kris Zyp. 2009. JSON Schema. http://json-schema.org/