Empirical Analysis of JSON Schema Use

Ammar Alsulami
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
afa7686@cs.rit.edu

***Abstract*—Coming from the wide adoption of JSON schema, this paper is devoted to investigating the use and characteristics of this technology. We collected, prepared, and analyzed 47,610 json files to draw meaningful conclusions for schema developers. Even with polishing of schema versions, version four is the most commonly used among users and string types outnumbers other types in terms of quantity. The majority of errors while validating schemas is due to using new features while refereeing to an older definition of the schema.**

***Index Terms*—JSON; Schema; Analysis**

I. Introduction

Java Script Object Notation (JSON) is a data exchanging format that plays a viral rule in developing and reshaping data exchanging methods. It was first introduced by [1], since then it is rapidly spreed to reach out and cover other applications such as storing data, sending data over the web (REST API) due to the flexibility and ease of use that JSON technology excel in. It is a key-value pairs document where each key represents a property name and all keys in JSON documents must be unique.

```
{"university": "RIT", "city": "Rochester"}
```

Listing 1: JSON example

The flexibility and expressive power of JSON is inherited by the nature of key-value pairs data structure where a JSON file user have the ability of structuring nested JSON document as they require. In that case the nested JSON document is called an object. Beside supporting complex data structures, JSON supports basic data types such as Integer, Numbers, Boolean, and Strings. With this flexibility and the potential complexity JSON documents has become one of the most successful data exchanging formats.

With the ever-increasing popularity of JSON as a data exchanging format, people start to worry about the validity and conformity of such data as well as the way of expressing the structure of their data to the corresponding users. One popular solution is to provide an example of the needed data and write the JSON document Validator for each application. However, that was not feasible in the scalalbitiy and the modularity aspects where each and any update to JSON schema structure requires an update to the validator itself, keeping in mind that the problem of expressing the data remains unsolved. The first ever attempt to address and formulate a solution to this problem was done by [2] that is called "Foundation of JSON Schema". JSON schema definition [3] is a set of rules and methods that defines the syntax and semantic of JSON schema in such a way that maintains JSON's flexibility and expressive power. JSON schema is also a valid JSON document, yet it is mainly used in validating JSON documents that written against. For example if an API "A" needs to validate a request from clinet "B" then API "A" should define a schema using the proposed JSON schema definition to validate documents that was received from client "B". The proposed schema definition has been adopted by large number of applications, as of July 3rd 2021, there are more than 87,000 JSON schema in GitHub that use the proposed definition and the newly born JSON schema has reached to mature state by releasing more than nine drafts of the definition over the recent few years.

```
{
    "$schema":"json-schema.org",
    "type":"object",
    "properties":{
        "city":{
            "type" :"string"

        },
        "university":{
            "type" :"string"

        },

    },
    "required":["city", "university"]
}
```

Listing 2: JSON schema example for listing 1

With the existence of a formal definition of JSON schema, the need of a real implementation of that definition had never been that vital before. There were several proposed programming solutions in different programming languages, but the one that excel above them is AJV validator[4]. It is the fastest among all JSON schema validator with clear documentation and high level of modularity. AJV validator provides a feature called **strict mode** from which the tool user have the option to manipulate the schema validation by adding or skipping steps from the validation process. By default, AJV

validator validate the entire schema starting from the structure of the schema until the user-written regular expressions, but with the help of the strict mode options the user can skip validating all these options including the validity of the schema itself not just what is inside the schema. Also, it allows schema users to add keywords that must be ignored during the validation process. These features collectively made AJV the optimal validator for our experiment.

The abundance of JSON schema requires an empirical analysis to investigate the efficacy of JSON schema features in terms of fulfilling the need of schema users. This was done on a small scale by [5] on almost 159 JSON schema documents. But these documents is very limited in numbers and do not reflect the entire community of JSON schema. In our work, we will try to expand the schema analysis both in numbers and features aspect in the process of capturing the entire space of schema users. However, we limit our work to just the formal definition of JSON schema.

The structure of the paper will as follows: section 2 will be devoted for methodoly, section 3 for the data collection stage, section 4 for the schema analysis, section 5 for discussing the result of the analysis, section 6 to explain the related work, and section 7 for conclusion and future work.

II. Methodology

Noted by [5], the large number of schemas was stored in Google BigQuery. Our work will be splited into two phases the first phase deals with the data collection steps and the second stage will deal with the schema analysis along with its supported keywords.

A. Data Collection

In the data collection phase we will start by collecting all available schemas in that use the standard JSON schema definition in Google BigQuery. Because we expect the result to have duplication and errors and the existence of these two types errors will lead to misleading result, we will remove any duplication in the data and validate all JSON schema before we carry our analysis.

B. Schema Analysis

In this phase we will analyze high level features starting with the distribution of schema versions then it will continue with help of AJV validator to validate these schemas under different settings. To gain more grasping details to the schema complexity we will measure the length of each JSON schema after pretty printing. After completing the high level features analysis, this study will shed the light into low level features analysis by examining the frequcny of supported type usage along with their supported keywords. Lastly we will finish the analysis by analyzing schema composition, annotation, and schema re using.

III. Data Collection

Data collection phase is crucial for the success of any project that involves sourcing and collecting raw data. Our project is no different, we utilized GitHub repositories as the data source from which will extract JSON schemas and prepare it by removing any duplication and ensuring their JSON validity for our analysis using BigQuery as the project data warehouse and python json library as the library of choice to validate JSON files.

A. BigQuery

BigQuery is a serverless data warehouse that allows its users to run large query over terabytes of data in timely efficient manner. Google BigQuery is compatible with ANSI SQL where it allows its users to run regular SQL queries over the desired tables to ease the learning curve for new users. One additional feature, which enables us to carry out our analysis, is that it includes public datasets from wide variety of organizations and data sources such as world-bank, GitHub, and crypto-currency etc.

B. GitHub activity dataset

In our project, we will use "GitHub activity dataset" stored in BigQuery to extract JSON schemas. As of July 3rd 2021 The dataset contains more than 3 million open source repositories representing a wide range of projects, the size of the data is over 3 terabyte containing more than 163 million files. Also, the dataset comprised of 9 tables, four as sample tables and the rest for the entire dataset. The project will focus on two tables of relevancy contents and files tables.

C. Data Extraction and JSON validation

We ran a query to extract all files that end with *".json"* and contain the url *"www.json-schema.org"*, the url refers to the standard definitions of all drafts, out of the BigQuery. The result of the query is the following: We had a CSV file that contains approximately 47,610 entries. After filtering out the duplication we were left with 30,012 unique entries that will be processed to ensure their JSON validity.

The JSON validation stage encompasses loading the files into python script that assures, using JSON library, each record validity before it has been considered for the analysis stage. The result of this stage was almost all the records, except 766, were valid JSON files. Those who failed the validation stage were manually analyzed to highlight the common mistakes among them. Interestingly, one major reason for failing the validation stage was using python dictionary's notation instead of JSON's in writing the presumably JSON files. Such error may occur because of the similarity between their notations, or serializing python dictionaries directly before converting them to JSON. Other notable reason is that few of the files contain double back slashes, JavaScript comments tag, which directly lead to invalid JSON document. The result of the JSON validation stage is 30,012 thousand valid JSON documents that will be used in the following analysis.

IV. Schema Analysis

A. Schema Distribution

After obtaining the dataset, the analysis will start by analyzing schemas' versions by looking at the top level *$schema*

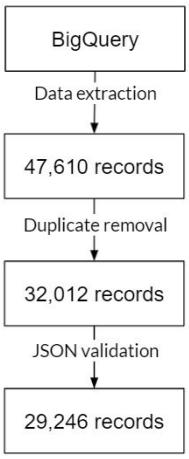

Figure 1. Data Collection steps and result

property. Almost 3,972 JSON files do not have top level *$scehma* keyword. That is due to few reasons such as the schema is a part of larger JSON file, or the targeted url *www.json-schema.org* was included for other reasons not including defining a JSON schema, these schema will be out of the scope of this analysis.

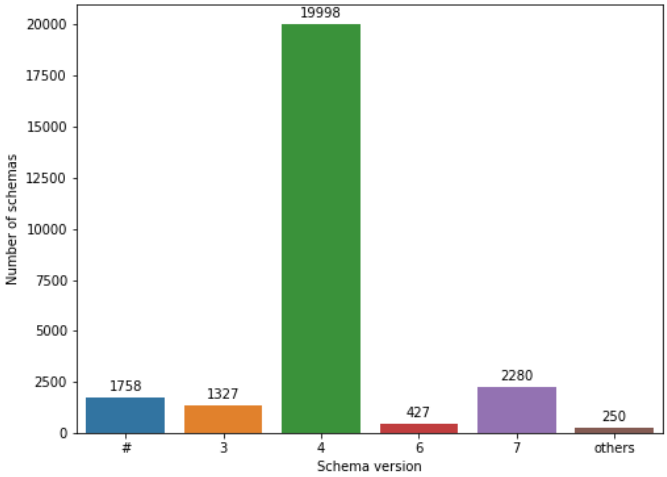

Figure 2. Schemas version distribution

As figure 2 shows, version 4 is the most widely used schema version, it is used more than all other versions combined. comes behind it is version 7, even though it is two years younger than version 4, it did not get the same adoption as its predecessor. One notable observation is that versions 1,2,5,2019 etc were not used as the other popular versions. they were rarely used, at least in open source projects, as an example version 2019 is present in less than 90 schemas.

B. Schema Validation

AJV validator [4] provides the option of validating a schema during the compilation stage in different modes. The variations of schema validation is considered a part of what is called strict mode. By default strict mode validate JSON schema during the compilation stage, and AJV user has the options to adjust schema validation to match their needs. In this analysis we sat all strict mode options to false and in each run we turned just one option to true to measure the impact of validating one option, such as *strict types* on the schema validation stage.

Strict mode option	Number of valid schemas
All false	14706
strict schema	7771
strict number	14706
strict types	12164
strict tuples	14187
allowDate	14706

Table I
SCHEMA VALIDATION RESULTS UNDER DIFFERENT SETTINGS

By examining the result, clearly that out of all schemas, when strict mode is on, two thirds of the them were invalid. Different behavior. The number of invalid schemas degrades significantly when strict schema option is turned off resulting in 14,706 compiled schmeas. Furthermore, allow date and strict number options has no effect on the compiling stage indicating that these two options have zero to minor presence across all JSON schemas. For invalid schemas, The errors were mainly due to unknown keywords or formats that are present in the schemas. Further discussion about the errors will be provided in the discussion section.

C. Lines of Code

When measuring the complexity and time needed to parse and validate JSON schema. Lines of code comes in handy to measure a schema complexity. It is also beneficiary for validators developers knowing such information to reevaluate JSON Schema validators' validation speed as in [6]. In the previously mentioned json-schema-benchmark the validation speed were based on test cases which are relatively smaller in size than the real JSON schemas.

Before measuring the schemas' lines of code each schema is readjusted using 4 spaces indentation to ensure that they share the same format for fair measuring.

As figure 3 displays that the mean length of a schema is 288 lines which is considerably larger than typical test cases provided by JSON schema test suite [7]. Interestingly, the largest schema contains 32,910 written by [8] used for computer graphics purposes. while the smallest schema contains just 3 lines representing an empty schema that just declares top level ***$schema*** key.

*D. **Data types***

In the next step of the analysis the light will be shed at the basic data types that are supported by JSON schema. String, Integer, number, boolean, null, arrays represent the

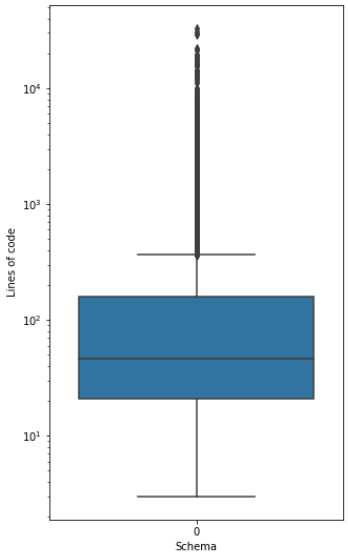

Figure 3. Schemas lines of code in log scale

basic supported types by JSON schema definition as well as other user defined types. This step of the analysis will focus on analyzing these types and their associated keywords. The analysis will be divided as follows: first, each occurrence of any of these data types will be counted. second each data type will be examined solely with its associated keywords which will be used later in the analysis to draw a typical JSON schema.

Type	Count
String	239,044
object	111,097
Integer	27,823
Number	30,489
Array	52,268
Boolean	31,444
Null	4,866

Table II
DATA TYPES FREQUENCY

1) Data types distribution: expectantly as table 2 expresses, string types is the most common types across JSON schema where it was defined more than all other types combined.The least used type is null types. the reason behind that it is mainly used in schema composition such as ***oneOf*** or ***anyOf*** where it is required to not have an empty values.

2) ***Strings****:* Analyzing string types with its associated keywords, ***minLength***, ***maxLength***, ***pattern*** and ***format*** comprised of counting the frequency of each usage of these data types along with the common patterns and format. such counting provides valuable details for the schema inventors where they can target and improve on those area by providing additional support for those who needs special format or features.

keyword	Count	% of strings
format	10,617	4.44%
pattern	21,440	8.96%
minLength	17,420	7.28%
maxLength	17,221	7.20%

Table III
STRING ASSOCIATED KEYS FREQUENCIES

In JSON schema ***format*** keyword refers to a set of built-in regular expressions that the schema definition supports natively. This set is defined to support and unify popular regular expressions. The majority of regular expressions that is supported by JSON schema is used in web applications environment such as **ipv4**, **ipv6**, and **uri**.

format	% of formats
date-time	39.58%
uri	27.36%
ipv4	3.95%
ipv6	3.61%
Duration	3.46%

Table IV
THE MOST FREQUENTLY USED FORMATS

Noticeably, duration format is not defined by the formal definition of JSON schema, yet it is the fifth most frequent used format by almost 367 declaration in more than 100 schemas.

To ensure flexibility and option diversity, schema definiton support defining regular expression in which a string entry must be valid against. This feature is what is refereed at as ***pattern***. Because of versatility of ***pattern*** in defining unlimited pattern of reguralr expressions including those which are defined under ***format*** category the, the popularity of ***pattern*** overruns other strings associated keywords especially ***format*** where ***pattern*** is almost as twice as it.

pattern	% of formats
`^(\s|\S)*$`	8.82$
`^([\w\/-]|[\w-][\w\/-]*[\w-])$`	7.56%
`^[A-Za-z0-9:_\-]+$`	6.95%
`^([a-z0-9-]+):([a-z0-9\.-]+)` `:([a-z0-9-]+)?:([a-z0-9-]+)$`	6.95%
`^[\w\/\.:-]+$`	5.53%

Table V
THE MOST FREQUENTLY USED PATTERNS

As table five displays, that the most commonly used pattern is a regular expression that accept anything. The reason behind using such pattern, which is equal to the absence of pattern key, is that it works as a placeholder by software-generated schemas where each string type entry contains pattern definition. The second most used pattern is a pattern that is valid against system paths expressing a reasonable need for adapting this regular expression in JSON schema's supported formats.

3) ***object****:* The second most used types in JSON schema is object type, which is the core foundation of JSON files itself from which it gained its flexibility and expressive power. The associated keywords with objects are oriented to the size

and the flexibility of the schema. Using object's keywords gives schemas' owners the ability to control not just the required entries, they also can control the degree of fredom that allows JSON to be viral. ***additionalProperties*** key accept boolean keywords that allows the owners of the schemas to control if they allow additional key-value paired into their system. Moreover, ***required*** targets the required keys that must be present in all JSON documents that conform to desired schema.

keyword	Count	% of objects
additionalProperties	42,688	38.42%
required	52,637	47.37%
minProperties	558	0.50%
maxProperties	248	0.22%

Table VI
OBJECT ASSOCIATED KEYS FREQUENCIES

Approximately 30.4% objects do not allow any additional features while the rest of objects allow additional features either by setting the ***additionalProperties*** to **True** or leaving it undefined indicating the same result as setting ***additionalProperties*** to **True**. This behavior implies that one third of the schema users go the extra mile of defining all features that their schmas should accept.

4) ***Numeric values****:* Numeric types, ***Integer*** and ***Number***, have always been an essential part of JSON. In JSON schema the use of numeric types becomes more important due to their ubiquity within the schemas. One average each schema contains approximately four definitions of numeric types, two for ***Integers*** and two for ***Numbers***. Building on the previous, an investigation of the use of numeric types is conducted to gain further insight of numeric types and their associated keywords. Furthermore, we resonate Analysing both ***Integer*** and ***Number*** types together due to the identically they share in associated keywords. The table below illustrates the frequency of each associated key word.

keyword	Count	% of numerics
mutlipleOf	365	0.62%
minimum	16,307	27.97%
maximum	8,296	14.22%
exclusiveMinimum	1,000	1.71%
exclusiveMaximum	241	0.41%

Table VII
NUMERIC TYPES ASSOCIATED KEYS FREQUENCIES

By examining the result, it become clear that associated keywords is widely used among ***Numeric*** types when it is compared to their counterparts in ***String***. where approximately 27% of numeric types definition use one or more associated keyword. Another noteworthy observation, 97% of the times that maximum value is defined, minimum value is defined too. the same phenomenon is extended to cover inclusiveMaximum and inclusiveMinimum with approximately the same percentage as well. However, the opposite inference can not be inferred, minimum presence is not linked with the presence maximum value by high confidence rate.

5) ***array****:* : Array types have wide variety of defining options when compared with other JSON schema nativly supported types. Beside defining the min and max length of the array, the user have the options of defining whether an array must be treated as a set by assigning **True** as a value of ***uniqueItems*** key. Moreover, what an array must contain in terms of type specific values or the order of assigning those types within the array is left as an option for the schema user too. One use case of controlling the order necessity is formed when defining an address needed by a schema. **[house number, street name, street type, city, state, zip]** is an example of address composition where the order of the entries within an array matters. The previously mentioned behavior can be controlled by the key ***items*** where the user have the option to define one type array or order specific array by assigning the needed value to ***items*** key. ***contains*** key adds another aspect to array definition where an array instance becomes valid if it contains one or more instances of the value of the key ***contain***

keyword	Count	% of arrays
uniqueItems	4,666	8.92%
contains	4	0.00%
items	51,007	97.59%
minLength	13,282	25.41%
maxLength	6,149	11.76%
additionalItems	1785	3.41%

Table VIII
ARRAY ASSOCIATED KEYS FREQUENCIES

Clearly that the use of the associated keywords is less frequent in array types than object types, even though they share a lot of definition features. One axiomatic reason is that, multiple of the use cases of these keywords can be interpreted by the values of ***items*** keyword which has been abundantly used when defining an array type. the aforementioned reason can be supported by the frequency of using **min** and **max** where it can not be fully specified by ***items***.

Nearly 8.8% of all defined array types are set to be set of unique items. meaning that roughly 91.2% of schemas have assigned the value **False** for ***uniqueItems*** key either by direct assignment or leaving it undefined. similar behavior of not fully controlling arrays is exhibited with ***additionalItems*** key where just 2.2% of all arrays instances disallow any additional items directly. Yet, further analysis needed in arrays to reveal and combine overlapping result between keys.

6) ***Annotation****:* When thinking of annotation it comes to mind tools and rules that is mainly defined to help and provide further information that is needed by users of a tool. In case of JSON schema, annotations provides the same benefits except the annotation is also a key-value paired within JSON documents that is intended to work as a guide and an explanatory keys for those who intended to write documents that must conform to a schema. Annotation keys can be divided into two main categories one affects the schema itself and declared to be used during compilation time ***const***: in case of constant entry,***default***: if a value of key is not present,***enum***:

to declare a finite set of value options from which a user must chose, and ***readOnly***, ***writeOnly***: to set the rules of APIs or documents user which value is intended to read and which to write. And the other category, that must be ignored by a validator during the compilation stage, comprised of ***title***, ***description***, ***$comments***, and ***example***.

keyword	Count
const	2,731
default	34,967
enum	67,108
readOnly	230
writeOnly	6
example	3,218
$comments	7,139
description	273,399
title	33,682

Table IX
ANNOTATION KEYS FREQUENCIES

By far among all annotation keywords ***description*** takes the lead in terms of usage, roughly 53% JSON shcema type-specific keywords have a description. The same can not be implied for ***title***, even when the number are almost double the number of schemas that we used for the analysis, nearly one third of JSON schema are using ***title*** because each JSON shcmea is not limited to use just one title. For example, One schema uses 1158 titles, 2299 descriptions, And there are several shcmeas that follow the same behavior. Backing to the category that affects the schema, with the large magnitude of ***enum*** types, they enclose interesting values and behaviors that overlap with other natively supported keywords. The first keywords is ***const***, almost 37130 enums are using single-value enum, the same effect as defining ***const***, instead of using ***const*** for better practice. However, by knowing that large number of JSON schema are generated by softwares and the same softwares that generates mutiple-values enum are suppose to generate single-value enum, the reason for this practice comes clear that it is more easier to write single piece of code that is responsible for all enums reading and writing instead of dividing the work into multiple pieces. The second keyword that is overlaped by enums is ***boolean***, multiple enums' arrays just contain **True** and **False** which could be easily, and for better practice, replaced by ***boolean*** values ,yet the same reason for the overlap in const is implied on this case too but the effect of the overlap is considerably lower than ***const***. Even though, ***readOnly*** and ***writeOnly*** are three years old and they were meant to mainly serve within API's environment, they have not been used considerably, it is even worse for ***writeOnly*** comparable to ***readOnly*** where only 5 schemas are declaring it. On the other side, the rest of annotations keywords have been used adequately. Furthermore, the second category of schema annotation keys, the one that works as a guide for schema reader, is used significantly. The previously mentioned conclusion, entails that even if schemas were generated by softwares these schema were intended to be read by users.

*E. **Reusing schema***

Ruse code has been always an integrated part of the majority of programming languages, JSON Schema allows similar practice that allows developer to reuse their schema or part of it by defining reference point using ***$ref*** keyword that points to one address under ***definition*** property. Using this feature unlocks new horizons for using JSON schema and it ensures easiness of defining and more robust schemas. It is safe to say to that reusing schemas is a common practice among schemas considering the nature that it was created for. Almost 34% of the collected schemas are reusing pieces of their schmeas with different frequencies.

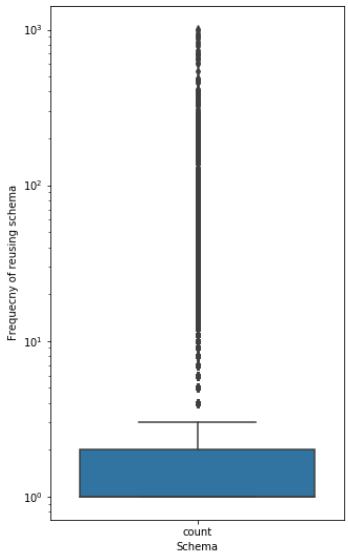

Figure 4. Reusing schema definition frequency

By examining figure 4, we can conclude that, in most cases, schema definitions are reused once, meaning they were defined under ***definition*** property then they were used in schemas. This practise could be interpreted in two ways either schema developers are planing to extending their schemas. or they are defining their schemas and then reuse to better organization.

F. Schema composition

Another feature that is provided by JSON schema definition is schema composition. It enables schema users using specific keywords to compose multiple schemas from multiple sources into just one schema. The definition of these keys is based on algebraic operators, ***allOf***, ***anyOf***, ***oneOf*** are meant to match **AND**, **OR**, and **XOR** characteristics respectively. Due to the speciality of using these operators, the presence of composed schema is slightly less than what features we have analyzed thus far, approximately 26% of JSON schemas contain one or more schema composition operator with significant advantage

for ***oneOf*** keyword compared to other schema composition keywords as figure 5 shows.

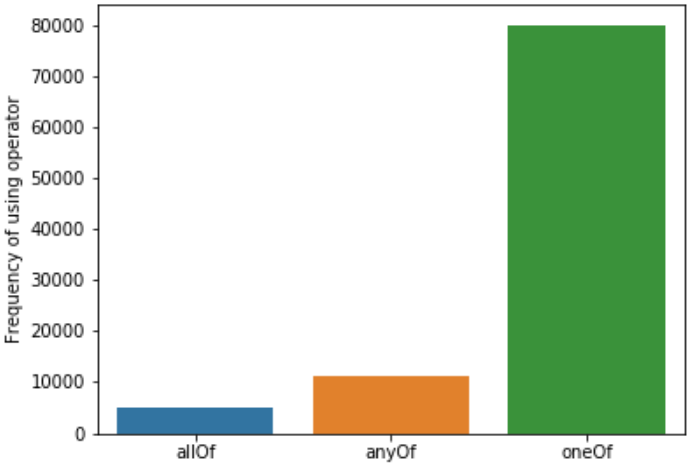

Figure 5. Schema compostion keywords

One possible reason for the lack of using ***allOf*** property is that it can be easily replaced by ***required*** feature under the definition of ***object*** types which is decently used in JSON schema. One supportive evidence of this finding can be derived from the frequency of ***oneOf*** when has no alternatives within the standard JSON schema definition which requires those who want to preform **XOR** function to use the only available feature that support this function ***oneOf*** key.

V. Discussion

During compiling stage, it was clear that we lost almost two thirds of schmeas when strict mode is on due to their invalidity. Losing this many of data arises an important question, What are the common mistakes among JSON schemas? The majority of errors were because unknown formats and unknown keywords. For both formats and keywords, the reason is not only these formats and keywords do not exist in JSON schema definition, in the majority, they exist but the reference of schema definition that these schemas refer to does not support these keywords yet. For example, ***readOnly*** keyword was the main reason for schamas invalidity for 745 schmeas, all of these schemas use version 4 definition which does not support ***readOnly*** property. The same reason for ***example*** where almost 254 of schemas that fail the validation is because they refer to version 4 instead of version 6 that supports this keyword. Actually, the non-presence of unknown keywords in schema definition was slighlty less than the present ones. In one case which is **"resourceDefinition"** which was introduced by Microsoft. **"resourceDefinition"** accounts for 935 schema errors in our data. However, that does not spare the AJV validator as one cause of errors. sadly, it does not support validation version 3 of the schema. Thus, they all failed the validation and we can not assume their validity while we are not able to compile them using the validator. On the other side of the spectrum, schema invalidity due to invalid syntax is rare among those who use the schema. Approximitly, 7.07% of the collected schema were invalid because of invalid syntax in the schema itself, highlighting a fact that most of schema users are ,at the very least, aware of one formal definition of JSON schema.

In general, JSON schemas show a humongous variation in size and feature use. where the sizes of schemas ranges from 3 to 32,910 lines. Almost the same variation is exhibited when looking the at JSON schema from every other aspect such as type distribution, association keyword, and schema compositions. One main reason of this variation is the applications where JSON schema are used in. They vary from simple application to infrastructure configuration.

It was mentioned multiple times in the analysis software-generated schemas which accounts to significant portion of the collected schemas. We mean by these the schemas, the class of schemas that was generated by softwares. Even though, there is no absolute certainty when deciding if a schema was created by a software or they are pure human work, we assume schemas that show certain behavior such as large length, using all associated keywords with all types, even if it was harmless to neglect them, or correct unintended use of schema features like ***enum*** with boolean array. While using software to generates schema is a brilliant idea, using such softwares have lead to new ways to instantiate schemas. In the positive side, they allow users to create very large schemas that would be infeasible for human to define and comprehend which allow the schema to be used in more complex applications. They are perfect to use when it comes to avoid typos and nesting schemas. On the negative side, in few occasions they did not use the schema as intended to be used. For example, using ***enum*** with boolean values, where they suppose to use boolean to ease the schema interpretation and allow JSON schema validators to function properly. On the side of easing the interpretation, instead of using natively supported formats, multiple software generated schemas are using regular expression to define predefined formats. In the end, further investigation needed for this type of schemas.

VI. Limitation and Future work

Despite the fact that our work is almost two order of magnitude of [5], we did not consider all JSON schemas available in BigQuery. Our work was limited to JSON files that use standard JSON schema definition. As of July 3rd 2021 there are almost 87,000 JSON documents that contain top level **$schema** keyword, Roughly double the files that use standard JSON schema definition. Most of these files use modified version of the schema definition which potentially would work as a great starting point to explore what the community of JSON schema needs to be adopted by later versions. Another limitation to our work is that AJV validator have a considerable false positive rate, almost 10%, in validating version 7% which would affect the result of this analysis. One solution for this dilemma is to use more than one JSON

schema validator to validate the schema before any another analysis. Lastly, in few occasions large shcemas skewed the result of the analysis. Using outlier elimination methods to eliminate these schemas would result in better approximation for what a real JSON schema look like.

VII. Related work

As of now, the only related paper to ours is [5]. Their work was inspired from [9]. They were the first to take JSON schema analysis into empirical analysis studies, yet the number of schema that they analyzed in their work does not reflect the whole set of JSON schemas. One different factor is that in their work they manually classified JSON schema into four different categories, but in our experiment we dealt with JSON schema from different prospective were we considered JSON schema as a whole set from which we will derive conclusion in the way of dividing JSON shema. We value their work for being the spark that ignite the flame of this study, as well as bridging the gap between schema users and definers.

VIII. Conclusion

With the emerge of the recently born JSON schema, many different applications have utilized as a tool that works as data definition language for their JSON documents. With the most mature JSON schema validator [4], in this study we collected and analyzed 26,033 JSON schemas. We found out that the majority of JSON schema that use the standard definition are using schema version 4. Even with the majority of the collected schemas being invalid we concluded that most of the mistakes were because of using an old schema versions. Moreover, the diversity of JSON schema features use is positively correlated with its length and that the array types are overrun other types in the use of association keywords. Yet the joureny of analyzing JSON schema features is still an open research and we recommend utilzing GitHub data to analyze all schemas not only those which use the standard definition. We also recommend microscopic analysis to ***enum***,***arrays***, and user-defined types to better understand and reshape the future of JSON schema.

Acknowledgment

I would sincerely express my gratitude to my professor and supervisor prof. Micheal Mior for the continues help and support that enriched my knowledge and guided me through the adventure of exploring new research area.

References

[1] T. Bray, "The javascript object notation (json) data interchange format," *RFC*, vol. 7158, pp. 1–16, 2014.

[2] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*, ser. WWW '16. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2016, p. 263–273. [Online]. Available: https://doi.org/10.1145/2872427.2883029

[3] M. Droettboom *et al.*, "Understanding json schema," *Available on: http://spacetelescope. github. io/understanding-jsonschema/UnderstandingJSONSchema. pdf (accessed on 14 April 2014)*, 2015.

[4] A. J. schema validator team, "Ajv," https://github.com/ajv-validator/ajv, 2021.

[5] B. Maiwald, B. Riedle, and S. Scherzinger, "What are real json schemas like?" in *Advances in Conceptual Modeling*, G. Guizzardi, F. Gailly, and R. Suzana Pitangueira Maciel, Eds. Cham: Springer International Publishing, 2019, pp. 95–105.

[6] A. Ebdrup, "json schema benchmark," https://github.com/ebdrup/json-schema-benchmark, 2021.

[7] J. S. O. team, "Json schema test suite," https://github.com/json-schema-org/JSON-Schema-Test-Suite, 2021.

[8] "Extensible 3d (x3d) graphics," https://www.web3d.org, accessed: 2021-07-03.

[9] B. Choi, "What are real dtds like?" *Technical Reports (CIS)*, 01 2002.