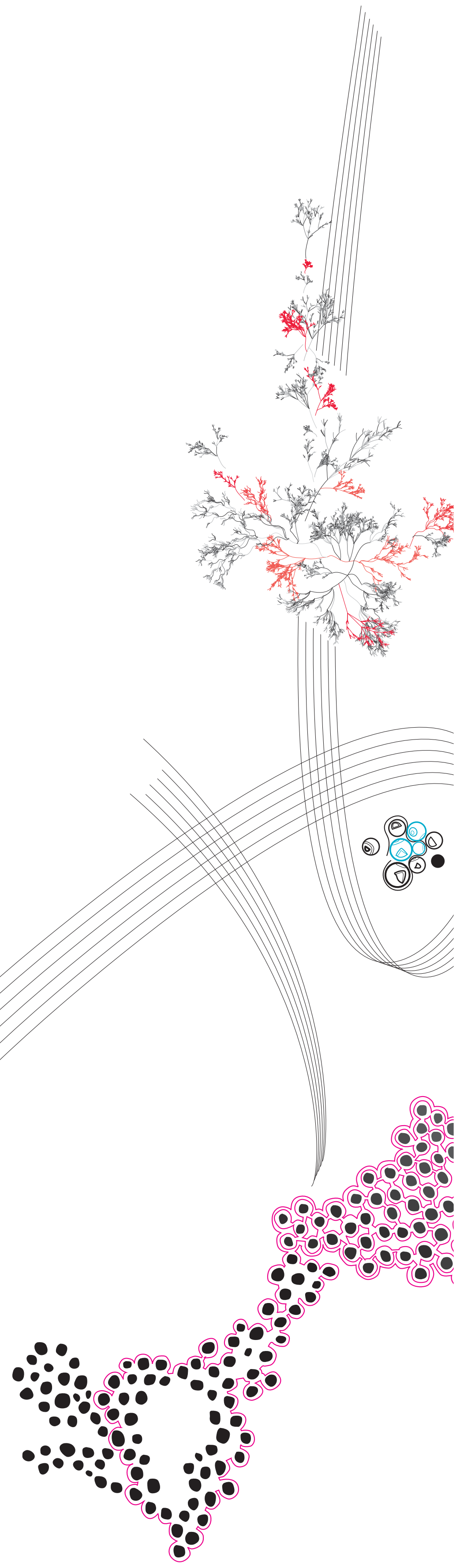
MSc Computer Science
Final Project



Incorporating User Inputs for Improved JSON Schema Inference

S. B. Broekhuis
s.b.broekhuis@student.utwente.nl

Supervisor: dr.ir. V. Zaytsev

December, 2023

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

JSON Schemas, as descriptive JSON files, define the expected structure of other JSON data, serving as a valuable resource for both developers and programs. They play a crucial role in data validation, testing, and maintaining data consistency. Since creating JSON Schemas can be challenging, it is common to derive schemas from example data. In this research, we focus on the introduction of user inputs during the inference process with the goal of reducing ambiguity and allow an algorithm to make, otherwise inconclusive, speculations from the sample data. We describe numerous strategies for utilising JSON Schema features based on sample JSON files and how they were implemented into a Kotlin program. We evaluated the program on five distinct real world sample JSON datasets from which the results showed it is able to infer complex patterns.

Keywords: JSON, Inference, JSON Schema, User Input, Interactivity

Contents

Chapter 1

Introduction

In today's data-driven world, the prevalence of JSON (JavaScript Object Notation) as a widely adopted data interchange format cannot be overstated. As JSON files have no specified structure themselves, JSON Schema offers a means to validate, test, and maintain the consistency of JSON data [10]. The JSON Schema specification has gone through multiple versions, each adding new features and changing existing ones. These regular revisions can make it challenging for users to construct schemas manually.

An algorithm can infer a structure from sample data, making it easier for users who may not fully comprehend the intricacies of the specification [1, 3, 6, 12, 20]. Although this approach saves time and effort, it may produce less complex structures that require further refinement. The samples hint at the conditions each field might have, and it should be possible for an inference algorithm to use these hints to build complex JSON Schemas. However, this can cause over-fitting.

To address this issue, our research introduces user inputs during the inference process. By doing so, we can reduce ambiguity and enable algorithms to make informed speculations that would otherwise be inconclusive. Users have a deeper understanding of the sample data, which can be leveraged to extract more information and improve the accuracy of the schema.

In this paper, we present seven interactive strategies for harnessing the capabilities of JSON Schemas, implemented in a Kotlin program. We evaluate our tool using five real world sample JSON datasets, highlighting its strengths and limitations.

This study aims to contribute to the field of JSON Schema inference, offering valuable insights into enhancing the precision and efficiency of schema creation, and the facilitation of data processing for a wide range of applications and industries. By reintroducing user inputs into the inference process, we not only aim to improve JSON Schemas inference, but also illuminate the potential for extending this concept to other domains.

Chapter 2

Background

The upcoming chapter provides context for the topic selected for this paper. To understand the process of inference, one must first grasp the need and concept of a schema, which in turn relies on comprehending the nature of the data that is stored.

Firstly, we clarify the difference between unstructured, structured, and semi-structured data. Secondly, we delve into the concept of schemas, encompassing their purpose in documentation, validation, and interaction control. Lastly, we explain the concept of *Informational Keys* and how it effects inference.

Readers who are already well-versed in these topics may choose to skip this chapter, focusing on JSON Schema inference and its applications.

2.1 Data

2.1.1 Notes and Spreadsheets

Since the dawn of writing we have been storing data. When we write/record/draw, we create data and information. Because the data and information is hidden behind abstract concepts such as language, extracting this information requires human input and this is unfavourable to humans. There is however a different method to create data. Data that can be accessed and processed easily without humans. We call this difference, unstructured and structured data. Unstructured data is information that is not organised in a defined matter. Data such as plain-texts, images, audio, are not defined by a structure and their information is thus difficult to process.

Example 1 Unstructured vs structured data

Emma (11) enjoys playing football and spends most of her free time practising with her team. Liam (9) is an avid reader and can often be found curled up with a good book. Aiden (13) loves to draw and is constantly creating new art pieces, often inspired by nature. Olivia (10) has a passion for science and spends her weekends conducting experiments in her backyard. Ethan (12) is an aspiring chef who enjoys experimenting with new recipes in the kitchen.

The text above contains fictional information about children and their hobbies. To extract the information with a program is difficult. Because the hobbies, names, and ages are hidden within language, such information would require language processing or machine learning to extract.

Name	Age	Hobby
Emma	11	football
Liam	9	reading
Aiden	13	drawing
Olivia	10	science
Ethan	12	cooking

In this table the same data is presented with columns of Name, Age, and Hobby. This data has less overall information, but provides an accessible format for programs to search through and process.

Example 1 shows why we stored data in tables or databases. However, tables do have limitation in their ability to store data. Their structure makes them inflexible compared to unstructured data, which can be anything. Data does not always fit inside a predefined structure. In the example it was trivial to convert the example of unstructured text into a table, requiring the storing of one name, one age, and one hobby per child. One might already realise that this causes a problem when children start developing more than one hobby. While it is possible to incorporate multiple hobbies by adding columns or tables that reference each other, it is apparent that we require a simple solution that bridges the gap between unstructured and structured data. Something, *semi-structured*.

2.1.2 JSON

JSON (JavaScript Object Notation) [9] is the file format this study will focus on. It was based on a subset of JavaScript and therefore commonly used in said language. It was "discovered" by Douglas Crockford around 2001 [25]. Since then, many standards were made. In Example 2 a JSON file is shown with data regarding children's hobbies. Here, Emma has more than one hobby. Where in Example 1 it was difficult to show multiple hobbies per child, in JSON it is possible and easy.

Example 2 JSON Hobbies

```
{
  "children" : [
    {
      "name": "Emma",
      "age": 11,
      "hobbies": ["football", "drawing"]
    },
    ...
  ]
}
```

In this example, a JSON file is shown with the data regarding children's hobbies. All data is stored via a key-value method. Here, hobbies is an array of multiple strings. This allows any number of hobbies per child.

However, what makes JSON particularly versatile is its language-independence, which allows it to be used seamlessly across a wide array of popular programming languages. Many popular languages have their own JSON parsing and generation libraries or modules. For instance, Python provides the 'json' module and Java has the 'Jackson' library [5, 17]. This universal adaptability makes JSON a versatile and widely accepted data interchange format, facilitating seamless communication and data exchange between different systems and platforms.

In addition to its widespread adoption as a data interchange format in various programming languages, JSON has found a significant role in the realm of NoSQL databases. NoSQL databases are designed to handle large volumes of unstructured or semi-structured data, making them particularly well-suited for modern, highly dynamic applications. The flexible and schema-less nature aligns seamlessly with the principles of NoSQL databases, facilitating the storage and retrieval of complex, nested data structures.

2.2 Schemas

Schemas are formal specifications that define the structure, content, and constraints of a data model. They provide a standard for describing and validating data, ensuring consistency and accuracy in its representation. Schemas can be used to define the structure and content of a variety of data formats, including JSON, XML, and others. For this paper, we will focus on only the data schema for JSON.

Example 3 An unclear JSON file

```
{
  "orderId" : "2022343-34AZEEF",
  "userId"  : 433,
  "reason" : 1
}
```

This JSON file is unclear as it does not describe itself. Questions may arise such as; What is `orderId`? Is `userId` required? Why is `reason` a number?
A schema would be able to answer these questions.

In Example 3, an ambiguous JSON file is presented. To describe it, we can utilise a JSON Schema [9]. A JSON Schema is a JSON file that outlines the characteristics, description, and data types of each field in the object, along with any additional conditions.

Example 4 A example of a JSON Schema.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "orderId": {
      "description": "Unique identifier of the order", "type": "string"
    },
    "userId": {
      "description": "Unique identifier of the user", "type": "string"
    },
    "reason": {
      "description": "Reason for the return", "type": "string"
    }
  },
  "required": ["orderId","userId","reason"]
}
```

In Example 4, we see a schema to describe the structure of the file in Example 3. Using this, it is clear that the `reason` field is invalid as it is required to be a string. The schema's structure is determined by the meta-schema located in the `$schema` field. Since there are different versions of the JSON Schema specification, this field is used to specify the version of the schema.

2.3 Informational Keys

As we have discussed, JSON is flexible due to its lack of structure. Due this flexible structure, the same information can be presented differently. *Informational Keys* demonstrate the significance of this. In a JSON file it is intended that the key is used to uniquely identify and retrieve a specific value from the data. However, it is possible to use the key as an identifier, attaching data into it. This often makes a file smaller, but results in inconsistent keys in the file structure.

Example 5 Informational Keys A

```
"people" : {
    "Alis" : {
      "age" : 34, "email" : "alis@example.com"
    }
}
...
"people" : [
  {
    "name": "Alis", "age": 34, "email" : "alis@example.com"
  }
]
```

In this example, two methods are shown to display a list of people. In the first example, the name of the person is used as an informational key. In the second example, an array of object are used.

In Example 5 an JSON is shown where information can be stored in the keys of an object. For a JSON Schema, this means that almost any key is allowed. As a result, an inference system will have trouble inferring this structure. The basic inference system would, for the second example, describe the structure of a person once, while for the first example it would describe it for each individual person. JSON Schema has the functionality to describe structure for any key that matches a specific pattern, however automating this is difficult. The system would need to detect the difference between normal keys, and keys that contain information.

Example 6 Informational Keys B

```
{
  "variants": {
    "powered=false": {
      "model": "minecraft:block/oak_pressure_plate"
    },
    "powered=true": {
      "model": "minecraft:block/oak_pressure_plate_down"
    }
  }
}
```

In the given example, we see a JSON configuration file for the "*blockstate*" specification for an oak pressure plate within the game Minecraft. This pressure plate is a block that has a `state` called `powered`, which changes when stepped on. The key `powered=true` in this situation serves as a condition for what model to display in the game when stepped on. Note that in Minecraft, blocks can contain various states, such as directionality, waterlogging, or connections to neighbouring blocks. These states can be combined by separating them with a ',' to create more complex conditions.

Example 6 shows a more complex real world example of how information can be stored in keys. A schema inference algorithm would, in this example, need to be able to parse keys and detect the regex pattern that corresponds to possible structure.

Chapter 3

Related Work

"That which is inferred; a truth or proposition drawn from another which is admitted or supposed to be true; a conclusion; a deduction."
from Wiktionary.

JSON, known for its structural simplicity, becomes more complex when JSON Schemas are involved because these schemas have a schema to follow themselves. To alleviate this complexity and facilitate schema creation, we can infer the schema directly from the JSON files. It automates schema generation from sample data by identifying the basic types (strings, numbers, booleans, objects, and arrays), patterns, and constraints, simplifying the process while ensuring data accuracy.

3.1 Research Papers

Since this project focuses on JSON Schema, this research is limited to JSON only. While XML Schema inference exists, the specifics are difficult to apply to JSON due to the difference in structure.

Much of the existing research focuses on the uses of inference regarding databases. The motivation stems from *NoSQL database*, where the lack of structure can result in the need of a schema. Analysing the existing works shows that JSON Schema inference is generally done in the following steps, often called *MapReduce*.

1. A large collection of JSON files is processed in parallel into a new format that the system uses.
2. Merging the collection of new formats into a single format. This process varies the most from different approaches.
3. The combined format is then transformed and outputted into a schema.

However,
In recent work, Čontoš and Svoboda [4] studied multiple current approaches for JSON inference and their limitations. They compared the works of Sevilla et al. [21], Klettke et al. [12], Baazizi et al. [1], Cánovas et al. [3], and Frozza et al. [6]. I will describe three of these studies that differ in their approach.

Inference with Graphs

Klette et al. [12] use a *Structure Identification Graph* to combine all the JSON properties from a NoSQL database into a single schema. The paper describes in detail how these graphs are created and combined into a single schema. As for its features, it is able to detect required and optional properties and union types, but foreign keys ($stop\_id \rightarrow id$) are not.

Inference with Equivalence relations

Bazazizi et al. [1] describe a different approach. They similarly use a MapReduce method, however the algorithm build two versions of the schema. The first version fuses all objects together with the same record. It marks all fields optional that are not present in all. The second version only combines if the record shares all the same fields. This results in a relatively small schema and a potentially large schema. Information regarding the combination of fields is kept in the large schema using this method.

The algorithm then uses a equivalence relation to determine if two objects are equal and should be merged into a single instance in the schema. This equivalence relation is a parameter for the algorithm. The paper continues with proofs that their algorithm is sound and accurate. While the study provides a valid and interesting approach to inferring a JSON Schema, I found the methodology and analysis to be quite complex and challenging to fully comprehend.

Inference with Traversal

Cánovas and Cabot [3] present an approach that generates class diagrams from JSON files, setting it apart from the majority of studies which predominantly concentrate on NoSQL databases. Their motivation comes from the need for a structure from services building or using APIs. This method traverses through the input JSON data, systematically crafting multiple class diagrams, which are then reduced into a singular class diagram. The creation, refining, and merging steps are documented and specified for each situation.

3.2 Online Tools

Besides research there are online tools available to infer a structure from a JSON sample or samples. This grey literature might or might not be open source and may infer from a single or multiple samples.

QuickType

QuickType [18] is a tool that is available as a website, program, library, and IDE extension written in TypeScript. It is able to infer JSON samples or a single JSON file a JSON Schema or other programming languages. On the website, the generated JSON Schema is, however, written with only definitions. Each definition is named after the file or folder of the sources. Since the resulting schema only contains definitions, this schema does not validate anything.

Saasquach's JSON Schema inferrer

The JSON Schema inferrer from Saasquach [20] is an advanced library written in Java. It is able to infer from multiple JSON samples a single JSON file. The resulting schema can be configured for different drafts, policies, formats, and more. The library has API features to expand the complexity of the inferrer.

Liquid Technologies & jsonschema.net

Lastly, Liquid Technologies [13] & JsonSchema.net [11] are both online JSON Schema generator tools that infer a JSON Schema from a single JSON sample. They both have limited options and settings and are not open source. They are easy to use compared to the other tools mentioned, making them useful when one needs a simple schema quickly.

Chapter 4

Project Description

This chapter offers an overview of the project, starting with the description of the problem. We will then outline the project's objectives and scope. To guide our research and approach, we will present a set of research questions and delve into our methodology for addressing these questions.

4.1 Problem Statement

The preceding chapter covered existing efforts in the realm of JSON Schema inference. However, the resulting schemas from these efforts tend to be relatively simple compared to the full range of capabilities of the JSON Schema specification. This limitation arises from assumptions these algorithms would be required to make.

Sample data, while informative about what is allowed, cannot convey what is disallowed. Consequently, any algorithm venturing into schema inference inevitably makes assumptions. A prevalent assumption involves defining the type of a field. For instance, if a field such as *foo* is always a number, the system deduces it to be exclusively numeric. This deduction rests on the assumption that, because we have not received any other type for this field, only numeric are allowed for the *foo* key. While this assumptions is trivial, it cannot be said for more complex situation.

Example 7 Fruits

```
{
    "fruit-type": "apple",
    ...
}
```

Consider the JSON snippet above with a `fruit-type` field that specifies the type of fruit referred to by the file. The number of allowable values for this field remains unknown at this moment.
Assume that we, as the inference system, are given 200 examples. We encounter five unique valid values of *fruit-type*, but can we conclude that these are the only five values that are valid? The input JSON files may not encompass all possible options. Thus, we can only infer that the minimum number of valid values for *fruit-type* is five.

In Example 7, we illustrate such an complex situation. When information is unavailable, the algorithm is unable to confidently infer the types of fruit. This underscores the need for exploring alternative approaches, such as a user-input-based method. In this context, users could play an active role by offering supplementary information or clarifying ambiguous situations.

In such instances, we can communicate to the user that, based on our observation of 200 examples, we have encountered only 5 unique values, potentially suggesting an enumeration type. An enumeration type is a list of all valid values, in this case, a list of valid fruits. The user may then verify whether the value indeed conforms to an enum type and accept the valid values.

4.2 Goal and Scope

As part of this research, our objective is to develop a JSON Schema inference program capable of handling such scenarios mentioned above, utilising a balance between under-approximation and over-approximation to ensure accuracy. We implement different strategies to handle specific scenarios for the user to respond to. These strategies will be described in Chapter 5.

This research describes the implementation of an inference program designed specifically for JSON files. The program will focus on the generation of JSON Schemas derived from JSON data. Certain aspects fall beyond the purview of this project, including the parsing of YAML files and the handling of NoSQL databases. The research maintains a dedicated focus on JSON Schemas and their inference from JSON data exclusively. To streamline the project and maintain a manageable scale, the research will employ a limited set of seven strategies for schema inference.

One notable exclusion from the strategies is the detection and handling of *informational keys* within JSON data. This exclusion is a consequence of the timing of this research. *Informational keys* became recognised as a concept during the evaluation of the implemented program. Because this concept is important to grasp, it was added to Chapter 2.

4.3 Research Questions

From the problem statement, we crafted these research questions.

Q1 What are the common types of JSON Schema inference algorithms, and how do they differ?

Q2 How can user input be integrated into JSON Schema inference algorithms?

Q3 How can user input refine JSON Schema inference algorithms?

4.4 Approach

In Chapter 3, we looked into the existing JSON Schema inference algorithms and tools. We concluded that most of the research focuses on extracting a structure from NoSQL databases. Besides the literature, we mentioned existing tools and libraries that can infer a JSON Schema. This displayed the current limitations in this topic and, with these insights, helped create strategies to improve the accuracy for JSON Schema inference algorithms.

This has answered research question **Q1**.

For research question **Q2**, we developed a tool that will be able to interact with the user. This program will have a GUI where the user can decline or accept speculations, altering data where needed. Seven different strategies will be developed to refine resulting schemas. These strategies are the first part of answering research question **Q3**. The second part is the evaluation, where we will run the created program on real world JSON data and the resulting schema will be evaluated in detail.

Chapter 5

Implementation & Development

This chapter presents a comprehensive account of the implementation of the Interactive Schema Inferrer (ISI). It includes design choices the implemented methods of interacting with a user. The source code for this project is available on GitHub [19].

5.1 System Design

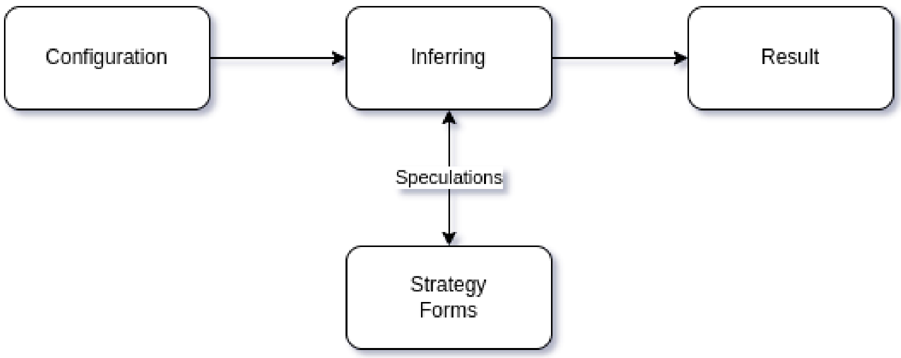


FIGURE 5.1: This figure illustrates the steps in the ISI. Where the Inferring steps often replaced by a strategy form requiring user interactions.

Figure 5.1 illustrates the operation of the system through three distinct steps:

The initial step involves displaying the *configuration view*, where the user is prompted to specify the schema version and select the JSON files to be used as samples. Additionally, a checkbox is provided to indicate whether the input JSON files are structured as an array, where each value in the array should be considered a sample.

The second step encompasses the inference process, in which the inferrer is constructed, providing it with all the strategies. A *strategy* is a method of improving an inferred schema by detecting speculations from a sample set, and using user input to confirm or deny *speculations*. It is crucial to emphasise that the absence of user input to affirm or reject these speculations would result in the generation of schemas overly tailored to the sample data. This is the underlying rationale why conventional inference systems are unable to incorporate such strategies. During the inference process, the strategies may replace the view with a form, enabling the user to response to a speculation. Upon completion, the *loading view* is reinstated and the response is processed.

Lastly, when the data has been processed and the inference has been completed, the *loading view* is replaced with the *result view*. This view presents the inferred JSON Schema as the outcome, along with a button for copying it to the clipboard.

5.2 Inference System

During the process of designing the system, it became increasingly apparent that the implementation bore striking resemblance to the work of Saasquatch [20], one of the online tools mentioned in Chapter 3. This inference system works by combining all the sample JSONs and traversing for each key all values provided, building up a schema from the bottom up. The library possessed the capability to build enum extractors and generic feature classes, which were essential components for implementing user interaction functionalities. Naturally, the decision was made to use the library rather than developing a new one from scratch.

However, the library was missing a crucial component to regarding user interaction. It was unable to provide context about the current field (key), as it only provided information about the values. If the system wanted to use user interaction, providing context to the user about what field needed clarification is crucial. To fix this, we opened a pull request to add the current JSON path to the API. Fortunately, the pull request was, after small adjustments, accepted. At the time of writing, a new version was not yet released, therefore there was no other option than to compiled the library manually instead.

5.3 User Interaction

To allow users to interact it is favoured to use a graphical user interface (GUI). Several options were considered for this purpose: *Compose Multiplatform* [8], which relies on Jetpack Compose for developing Android applications; *TornadoFX* [23], a Kotlin-based JavaFX framework; and *SurveyJS* [22], a JavaScript library for building forms.

Among these alternatives, TornadoFX emerged as the most suitable choice. Compose Multiplatform allows for the creation of visually sophisticated GUIs tailored for high-end applications compatible with multiple platforms. However, its implementation complexity surpasses that of the other options. SurveyJS, although visually appealing, is a JavaScript library that necessitates the setup of a local server to facilitate data transmission between the form and a browser instance. In contrast, TornadoFX allows for the development of a rudimentary UI application with minimal effort. Despite being no longer actively maintained, it continues to function well and has comprehensive documentation. Therefore, it was ultimately the chosen library for this project.

5.4 User Input Strategies

Each strategy is a class which implements a method called by the inferrer for each applicable field. Generally, it receives the following information to infer from; the preliminary schema for this field, the type of the current field (`array, number, object, ...`), the draft version provided, the samples of this field, and the JSON path of this field.

As mentioned in Section 2.2, JSON Schema has multiple versions, called drafts. These specify what keywords are available and how they should be used. Each strategy might be disabled or behave differently based on the version. In the following paragraphs, when referring to "all drafts", this encompasses all JSON Schema drafts from 4 onward.

5.4.1 Constants

A `const` is a keyword that specifies that a field is always this specific value. This keyword is available since draft 6 and is part of the validation vocabulary. When samples of a field consists of only a single distinct value the system speculates that this field is a `const`. However, this approach proves inadequate when confronted with limited sample sizes or, even more disadvantageously, when the sample size is merely one. In the latter case, the system refrains from making any speculations altogether.

5.4.2 Enumerators

A `enum` is a keyword that specifies that a field is restricted to a specific set of values. This keyword is available in all drafts and is part of the validation vocabulary. The system speculates similarly to the const. The system perform a division of the distinct sample size by the total size and examines whether this surpasses a predefined threshold. The determination of the threshold value emerged during testing, and led to value of 0.2. This threshold was selected to strike a balance between minimising false positives and maximising true negatives. It is essential to understand that the exact threshold value, is not a critical determinant in the scope of this project. The primary objective is to achieve reasonable coverage rather than pinpoint accuracy. Fine-tuning this threshold can be a topic for discussion and adjustment in future iterations.

5.4.3 Default

The `default` annotation keyword specifies that *"...if a value is missing, then the value is semantically the same as if the value was present with the default value"*. This keyword is available in all drafts and is part of the meta-data vocabulary. This strategy is unique in the sense that it does not influence validation of a JSON file. Nevertheless, it remains feasible and beneficial to deduce a default value. Initially, the process of speculating whether a field possesses a default value requires an analysis of the frequency distribution of distinct values within the sampled data. The system would employ the empirical rule to identify potential outliers in these frequencies. If such outliers are present, the system postulates that the most substantial outlier represents the default value.

However, through experimentation, it became evident that the effectiveness of outlier detection was not as reasonable as initially presumed. To illustrate this point, consider a scenario in which one value occurs 800 times while another occurs only once. In such a case, traditional outlier detection methods fail to identify the latter value as an outlier, as they tend to assume an average frequency of around 400. Consequently, a more straightforward approach was proposed.

This approach involves assessing the frequency of each distinct value and determining if the most frequent value appears in more than 80% of the cases. The threshold of 80% was chosen somewhat arbitrarily, but it seemed suitable during testing. Similarly to the enum strategy, the exact threshold value, whether it is 75%, 80%, or 85%, is not of critical importance. It should be noted that if the frequency is 100%, we assume it to be a constant and do not process this value further.

5.4.4 Uniqueness

The `uniqueItems` keyword specifies that an array field can or cannot contain the same value multiple times. This keyword is available in all drafts and is part of the applicator vocabulary. By analysing the array values of an field, we can speculate if the field can be marked with `uniqueItems` when each sample of type array for a specific field does not contain the same value twice.

5.4.5 Contains/PrefixItems

The `contains` (Draft 6+) and `prefixItems` (All drafts) keywords specify that an array should contain the a specific set of values, where `prefixItems` also specifies the index. This approach is particularly effective when applied in the context of post-order traversal, as it benefits from the prior inference of the schema beneath the current stage.

We use the preliminary schema to test whether the array always contains a specific condition. This strategy is exclusively used when the schema encompasses multiple conditions, typically in the form of an `anyOf` and/or `"type": [...]`. If a consistent pattern emerges where the same index consistently adheres to the same condition, the system designates it as a `prefixItems`. In the event that a user declines a `prefixItems` inference, the program will ask whether it should be considered as a `contains` instead. Here the the system provides options for `minContains` and `maxContains`.

5.4.6 MultipleOf

The multipleOf keyword specifies that an numerical value should be a multiple of a given positive number. This keyword is available in all drafts and is part of the validation vocabulary. By finding the greatest common divider (GCD) of the samples, we can speculate if the field can be marked with `multipleOf`. This only happens if the sample size and the GCD are both larger than 1.

5.4.7 Length

The last strategy implements keywords regarding the size or length of values. JSON Schema can add these conditions for Numbers (Range), Arrays (Item Count), Strings (Length), and Objects (Property Count). These keywords are available in all drafts.

This strategy waits until the inference is complete before asking the user for input. By doing so, the system can present the user with a list of all options at once (disabled by default), rather than multiple screens. During the inference process, the system keeps track of the minimum and maximum values for each condition mentioned earlier. This information is used to ensure that the user cannot set an invalid minimum or maximum value that would invalidate the samples. Additionally, for numbers the system provides an option to specify if the range is exclusive or inclusive.

Chapter 6

Evaluation

The previous chapter described the development of the ISI. This chapter will continue with evaluating the tool by executing it on specific datasets and examining their results.

6.1 Method

The evaluation procedure is as follows: The tool is executed on a designated dataset, and the resulting schema is reviewed. During the inference noticeable speculations or lack of are documented. Certain datasets originate from sources that already provide a JSON Schema, and in such instances, a comparison will be conducted between the derived schema and the source schema. The ISI serves as an extension of an established library, albeit with a distinct configuration where certain pre-existing features remain disabled intentionally. The deliberate omission of these features allows for the focus on newly added functionalities. Resulting schemas will, for example, not contain any format — strings with specific format rules, such as emails — inferences.

In the previous chapter we have mentioned the use of specific sample files for experimentation and system testing purposes. In the interest of preserving the impartiality of the evaluation process, it is important to abstain from including these sample files during the evaluation, as the software's performance has likely been optimised to align with them.

The following datasets will be used during the evaluation:

- Minecraft Biomes. [15]
- Earthquakes data. [24]
- NPM packages configurations. [1]
- IMDb movies example dataset. [7]
- OSI Licences. [16]

The resulting schemas will be available on the GitHub page as they are too large to provide in this paper.

[1]Extracted from public GitHub repositories.

The selection of these five specific JSON datasets for the study was guided by several considerations:

- **Real-World Examples:** The datasets chosen are grounded in real-world scenarios, providing a practical foundation for the study. This decision was motivated by the intention to ensure that the schemas inferred are relevant and applicable in genuine operational contexts. The authenticity of these datasets contributes to the robustness of the study outcomes.
- **Diverse Use Cases:** One key criterion for selection was the diversity in the utilisation of the datasets. The chosen datasets represent a spectrum of applications, ranging from configurations files to scientific research data and database information. This deliberate variation in use cases aims to expose the inference algorithms to a wide array of JSON structures.
- **Variety in Data Types:** The datasets exhibit significant differences not only in their use cases but also in the types of data they encapsulate. This intentional diversity encompasses various data structures, field types, and nesting levels. This breadth in data types serves to challenge the inference algorithms and ensures that the resulting schemas are capable of accommodating a broad range of JSON structures.
- **Study Scope and Manageability:** The decision to limit the study to five datasets was deliberate, stemming from a balance between comprehensiveness and practicality. A more extensive dataset collection might not necessarily yield significantly different insights and could potentially overlap with the characteristics of other samples. By constraining the dataset count, the study aims to reduce the work while still ensuring a meaningful and focused exploration of JSON schema inference.

6.2 Samples

Sample 1: Minecraft Biomes

The first sample data that will be used is data from the game Minecraft. Minecraft is a video game made set in a world of cubes. A **biome** is a region in that world with its own geographical features and properties. A biome can have different grass, foliage, sky, water colours. Such information is stored as JSON files within the games files.

Notes & Comparisons

The unofficial Minecraft Wiki [14] describes the structure for custom biomes. This documentation is used to compare the resulting schema.

The initial point of distinction lies in the lack of fields within the `particle.options` object. Within the context of the game, certain biomes feature ambient particles that traverse the screen. In the case of sample biomes, these particles are defined through an `id` and a `probability` parameter. However, it is important to note that the game provides more intricate customisation options for biomes created by third-party developers. As these customised options are not utilised in the provided samples, they are consequently absent from the resulting schema.

Another notable result of the schema was the detection of default values for `fog_color` and `water_fog_color`. These attributes dictate, as a number, the colour of fog both within and outside of water. The system has detected for the `fog_color` the value 12638463 🟦 predominates, being employed in over 80% of instances. The inclusion of this information as a default setting will prove advantageous for third-party developers seeking to employ a standard fog colour in their biome implementations.

The complexity increases for the `temperature_modifier` field, which is an optional key. This particular field can assume one of two values: *none* or *frozen*, with *none* being the default in cases where it is omitted. Ideally, this field should be categorised as an enumeration encompassing these specific values. However, a challenge arises due to its optional nature. Since no JSON file would explicitly denote *none* in this context, the samples featuring this field consistently exhibit the only other option, *frozen*. Consequently, the system has mistakenly identified it as a constant value.

Lastly, we turn our attention to the `spawners` field, which delineates the entities that can potentially spawn within the confines of the biome. Each *mob category* field has the same structure, where the category is *monster*, *creature*, *ambient*, *water_ creature*, *underground_ water_ creature*, *water_ ambient*, *misc*, or *axolotls*. Ideally, the `propertyNames` keyword, in conjunction with `additionalProperties`, should be used to establish a consistent structure encompassing all mob categories without having to repeat the structure in the schema. However, due to the system inferring each field independently without considering other related fields, it fails to recognise the shared structure among these fields. This gives rise to two primary issues: first, the resulting schema redundantly represents the structure multiple times, and second, users are required to provide repetitive responses to identical speculations. Which provides opportunity for inconsistencies in user input.

Sample 2: Earthquakes

The second dataset in this study comprises GeoJSON[2] features representing earthquake locations from the past 30 days, sourced from the United States Geological Survey. This dataset, initially presented as a GeoJSON `FeatureCollection`, has been streamlined to exclusively include the individual *Features* arranged within an array structure. One might realise that this will change the resulting structure.

The proposed schema's architecture will be compared against the official documentation provided by the United States Geological Survey, as published on their website.

Notes & Comparisons

The resulting schema was of good quality, as it was able to detect all conditions accurately. All properties were detected, and marked as required. Because the resulting samples only contained *Features*, the `type` field was detected as an constant. In cases where data was missing, the samples provided a `null` value. This resulted in the schema allowing both for these properties. Nonetheless, it is worth noting that certain values were consistently featured in the data, and as such, the schema did not add the option for `null` to allowed. It is unclear from the documentation which values are or are not allowed to be `null`.

Delving into the specifics of the properties, we encounter noteworthy detection for enums:

- The `status` property astutely discerns whether an event has undergone human review, signifying this via the `automatic` or `reviewed` options. Notably, the `deleted` alternative, while mentioned in documentation, is understandably absent from the samples (and thus also the resulting schema).
- The `alert` property informs the alert level according to the PAGER earthquake impact scale, and was detected as an enumeration of `green`, `yellow`, and `orange`. The absence of the `red` sample came from the apparent lack of `red` cases within the last 30 days in the sample data — perhaps a fortunate twist of fate.
- The `tsunami` property, denoting whether an event occurred in an oceanic region, was correctly identified as an enumeration of either 1 or 0. It raises the question of why a boolean data type was not employed for this purpose. Possibly, it was the result of how booleans are stored in their database.
- The `type` property, categorising the seismic event, was detected as an enumeration of `earthquake`, `quarry blast`, `explosion`, `ice quake`, and `other event`. However, the official documentation does not specify this property as an enumeration. This detection leaves me uncertain whether this detection should be interpreted as a positive or negative result, as `other event` implies that the given options would suffice as an enum type.

Finally, the schema's length strategy allowed the addition of `minItems` and `maxItems` for the `coordinates` array. This would require the array to be comprised of three values (longitude, latitude, depth).

[2]GeoJSON is a format designed for representing geographical locations in JSON

Sample 3: NPM Packages

The next dataset contained samples for the JavaScript package manager NPM. Information about a package is stored in the *package.json* file present in each project. This file provides information about the name of the project, mark what dependencies that are used, macros to run scripts, and other configurations. The gathering of this sample data was done with the use of the GitHub API. Using this API, *package.json* files from public repositories were extracted and aggregated into a single JSON file. Due to the presence of potentially sensitive or personal information within this document, despite its publicly accessible nature, we shall refrain from providing it.

Notes & Comparisons

The NPM *package.json* file presents a formidable challenge for schema inferences. As mentioned in Section 2.3, when a JSON files use informational keys, inference becomes difficult. Ideally, a single definition would be presented in the `additionalProperties`. Unfortunately, the current inference system is not implemented to detect such usage of keys, treating each field independently. As a consequence, the system produced an exceedingly extensive JSON Schema, where each field, be it a library, dependency, script, or configuration choice, is specified. Comparing this to the version available on SchemaStore.org, resulting schema is appalling.

However, it does reveal numerous licenses to be an enum, which the other schema also specifies. The SchemaStore.org variant adopts, however, the elegant approach by utilising the enumeration an suggestion, permitting any string value while still documenting the most prevalent licenses through the use of the `anyOf` keyword.

Sample 4: IMDb Movies

The Internet Movie Database (IMDb) is an online repository dedicated to entertainment media. Its API documentation includes a curated dataset comprising JSON responses spanning a range of queries as an example responses. Among these queries, 'title with parameters' movie responses were specifically extracted and utilised as the primary sample data.

Notes

The sample dataset appears to be curated, as they predominantly featuring highly-rated films. This was apparent in the program's repeated speculation for ratings (such as IMDb and Rotten Tomatoes ratings) to marked as an enum. Interestingly, when dealing with ratings, as they are stored as strings, the inference system can therefore not infer a potential `multipleOf` constraint for ratings. Moreover, a substantial portion of the movies in the dataset are English, which suggests a bias in the samples. Consequently, the program erroneously assumed that `language` was a constant, a speculation I declined.

The program was able to detect Language, Genre, and Country as enums. A deeper understanding of the back-end infrastructure could enable more informed judgements regarding the enumeration of these attributes. For instance, if IMDb merely stores languages as key/string pairs. Noticeably, the language data is stored as an object with two fields: *key* and *value*, where the two fields were always the same. My assumption is that value would be different in other languages. If this were the case, an enum would be rather complex to implement. While it was chosen to designate them as enums, this choice notably inflates the schema's size.

The program identified a comical repetition in the lists of people, particularly in the cast and crew context. In the *fullCast* section, the program noticed a pattern regarding job descriptions that were listed alongside individuals. This field was also observed in specific job sections, such as directors, where all directors were specified as *director*. This keen observation caused the system to detect that field as a possible constant.

Sample 5: OSI licenses

The Open Source Initiative (OSI) is a organisation dedicated to promoting and safeguarding the rules of open-source software development. It maintains a comprehensive dataset of open-source licenses that developers can use for their software. This dataset is in the form of a JSON file, which will be used as the last sample to test the system on. Unfortunately, we were unable to locate a schema for this file to use for comparison.

Notes

Each licence has an array of identifiers that display the identifier of the licence in at most 3 different formats (SPDX, Trove, or DEP5). The system was able to detect the 3 types of format were as a possible enum. In the `text` field of the samples, the JSON file specifies the link to the licence and the type of the file. In this field the `media_type` property was correctly categorised into three distinct enumerations: *text/html*, *text/plain*, and *application/pdf*. Similarly, the `title` property was categorised into three enumerations as well: *HTML*, *Plain Text* and *PDF*.

However, it is noteworthy that the program did not inherently establish a direct correlation between the `media_type` and `title` properties, even though a clear correlation exists. As said before, the system processes each field independently, and is therefore lacking functionality in detecting correlations between fields. For instance, when `media_type` is identified as *text/html*, the corresponding `title` is *HTML*. This, and other previously mentioned, lack of correlation recognition highlights a potential area for potential enhancement in the program's functionality, as it could improve the accuracy of the resulting schema.

Chapter 7

Discussion and Conclusion

7.1 Research Questions Revisited

In Section 4.3, we have defined the following research questions to help guide the project.

Q1 What are the common types of JSON Schema inference algorithms, and how do they differ?

Q2 How can user input be integrated into JSON Schema inference algorithms?

Q3 How can user input refine JSON Schema inference algorithms?

Q1 Schema inference algorithms

We answer this question in Chapter 3, where we discussed existing inference algorithms. From the existing JSON Schema inference algorithms we found that most focus on generating a structure from NoSQL databases [1, 2, 4, 6, 21]. Generally, these inference algorithms use the *MapReduce* method for processing the files. Unfortunately, these algorithms often do not produce a JSON Schema, rather they produce class diagrams, or their own defined structure definition. Besides research there are online tools available to infer a structure from a JSON sample or samples. One of these was the work of Saasquatch [20]. We used this library in our tool was ultimately the basis for the implemented system.

Q2 User interaction integration

We addressed this question in Section 5.3, which explored the various ways of implementing user interaction. Users are prompted by the tool when it requires clarification, this halts the program until it receives an answer. This can happen during the inference or after the inference has completed. It was chosen to use TornadoFX as the library for the GUI as it was the best solution from possible options. During the inference, the inference system creates a basic schema from the primitive types of each file. It then calls strategies for each field with relevant information to improve the resulting schema. If the strategy thinks it has found an improvement, it asks and waits for the user to respond. To improve the user experience, the design of the UI for each strategy focuses on making it simple for the user to decline any speculation. Additionally, for strategies that would otherwise always require user input, they are instead combined and asked at the end of the inference process.

Q3 How users improve inference

In order to address this question, we have provided a detailed account of the strategies we have implemented in Section 5.4. These strategies were formulated by examining all the keywords in the JSON Schema and contemplating how a program could identify situations in a set of JSON files where a keyword would be appropriate. As JSON Schema can be expanded with custom vocabularies, there is no limit to the potential for other strategies.

As seen in Chapter 6, there were still some limitations in the inference process. In that chapter we evaluated the created program on five distinct JSON datasets. The evaluation of the five samples revealed a spectrum of quality, ranging from schemas deemed highly favourable to those considered significantly unfavourable from my subjective standpoint, reflecting the varying degrees of refinement that would be needed. We saw that the enumeration strategy was the most successful, improving the structure for almost all samples. In the following section of this chapter we will delve into the limitations and future work of this study in more depth. Nevertheless, the implemented strategies have demonstrated how they can assist a user in enhancing the resulting schema of an inference algorithm.

7.2 Reflection & Future Work

In the course of this study, it became apparent that JSON Schema possesses a far greater degree of complexity than foreseen. This complexity allowed for many strategies to be created, although many its features were not implemented. I recognise that, in retrospect, greater confidence in my abilities may have led to the implementation of additional strategies that could have further enhanced the system. In particular, strategies that would organise and structure parts of the schema. However, as the system's ability to detect structure improves, the task of organising specific schema components with similar structures becomes progressively more intricate.

Besides validation rules, a JSON Schema provides tools for documentations. The current system does not make use of these tools as it is hard to infer documentation from samples. Unless artificial intelligence is used, it is debatable whether this system could provide this functionality. Future systems might allow, similarly to how the length strategy works, to provide an end screen where all fields could be provided with an description. Alternatively — as the resulting schema is expected to be tweaked or altered — the system could provide empty description fields for users to fill in later.

A important part of this study was the challenge of striking a balance between user interaction and automated inference. As a system that is supposed to make the creation of a JSON Schema easier, excessive user involvement counteracts this. The design attempted to minimise the user interaction, where most of the interaction is required when *confirm* any speculation. Regrettably, within the scope of this study, I was unable to address this problem directly, highlighting the necessity for further exploration and refinement.

When we focus us on specific strategies, there are improvements to be made. Consider the strategy for detecting enumerations. This strategy was the inspiration for this study. The current implementation works well, however a different approach can improve it. Instead of separating the constant and enumeration strategies, they could be merged. Since an enum with a single value is equivalent to a constant, the system could easily replace it. An additional improvement would be to allow the user to specify if these values are suggestions. If so, the schema would wrap the result around an `anyOf` with the primitive type. This allows all values to be valid but still provides suggestions for autocompletion.

One might have noticed in the evaluation that the strategy for speculating prefix items and contains keywords was not detected in any of the samples. This might indicate that this strategy is not working as expected. The strategy works by testing the already made schema for the current field, but this sub-schema might be to complex to detect any consistent patterns. To avoid any influence of the sample data on the code, the experiment was conducted in a one-time manner. While this approach aligns with the objective of unbiased analysis, it does introduce a challenge when identifying potential issues or errors in the code post-experiment. Therefore, I chose not to attempt to find a solution for the strategy at this time. Future work should evaluate the implementation of this strategy.

Furthermore, the current `multipleOf` strategy can also be improved. At the moment, largest value is the only option to select. The option to select a common divider that is smaller than the greatest common divider would be a good quality of life feature. Additionally, as we saw in Chapter 6, some numbers are displayed as strings. Future work can improve this strategy to attempt to parse strings as numbers and attempt to detect a GCD in the string. The `multipleOf` keyword, however, does not apply to strings. For the system to specify this condition, we instead would use a regex on the string.

Continuing, we realised that sample do not always provide a complete depiction of a structure. We observed scenarios, such as Minecraft biomes, where programs or systems receiving JSON files would use a default value when a field is not present. This reveals that even the most advanced systems can not infer all nuances, and thus needs to remain flexible.

As we have discussed, informational keys are difficult to infer, but perhaps not impossible. Many JSON files we encountered prioritised human readability over adhering to the expected structure of a JSON file. For example, dependencies inside a NPM package could have been an array of objects with the same structure, where the name of the package would be value of a field. Instead, it was chosen to use an object, with the key the name of the package. Since keys are unique, this makes it clear when there exists a duplicate dependency. It is important to recognise that the complexity of informational keys can vary widely. It can be as basic as an enum of keys or as sophisticated as Minecraft Blockstate (see Example 6).

Besides these limitations, I am pleased with the results of this study. It showed how a complex JSON Schema can be created with the help of a user, in particular, enumeration. This research journey has not only contributed to my understanding of JSON Schema inference and but has also yielded valuable insights and lessons that can guide future work in this domain.

Bibliography

[1] Mohamed-Amine Baazizi et al. “Parametric schema inference for massive JSON datasets”. In: *The VLDB Journal* 28.4 (Aug. 1, 2019), pp. 497–521. ISSN: 0949-877X. DOI: 10.1007/s00778-018-0532-7. URL: https://doi.org/10.1007/s00778-018-0532-7 (visited on 01/30/2023).

[2] Mohamed-Amine Baazizi et al. “Schema Inference for Massive JSON Datasets”. In: OpenProceedings.org, 2017. DOI: 10.5441/002/EDBT.2017.21. URL: https://openproceedings.org/2017/conf/edbt/paper-62.pdf (visited on 12/20/2022).

[3] Javier Luis Cánovas Izquierdo and Jordi Cabot. “Discovering Implicit Schemas in JSON Data”. In: *Web Engineering*. Ed. by Florian Daniel, Peter Dolog, and Qing Li. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 68–83. ISBN: 978-3-642-39200-9. DOI: 10.1007/978-3-642-39200-9_8.

[4] Pavel Čontoš and Martin Svoboda. “JSON Schema Inference Approaches”. In: *Advances in Conceptual Modeling*. Ed. by Georg Grossmann and Sudha Ram. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 173–183. ISBN: 978-3-030-65847-2. DOI: 10.1007/978-3-030-65847-2_16.

[5] FasterXML. *Jackson Project Home @github*. 2023. URL: https://github.com/FasterXML/jackson (visited on 10/26/2023).

[6] Angelo Augusto Frozza, Ronaldo dos Santos Mello, and Felipe de Souza da Costa. “An Approach for Schema Extraction of JSON and Extended JSON Document Collections”. In: *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. 2018 IEEE International Conference on Information Reuse and Integration (IRI). July 2018, pp. 356–363. DOI: 10.1109/IRI.2018.00060.

[7] IMDb. *IMDb Sample Data*. Sept. 5, 2023. URL: https://imdb-api.com/API.

[8] JetBrains. *Compose Multiplatform UI Framework | JetBrains*. JetBrains: Developer Tools for Professionals and Teams. URL: https://www.jetbrains.com/lp/compose (visited on 10/26/2023).

[9] *JSON*. URL: https://www.json.org/json-en.html (visited on 10/26/2023).

[10] JSON Schema Working Group. *JSON Schema*. URL: https://json-schema.org/ (visited on 10/26/2023).

[11] JSONschema.net. *JSON Schema Generator*. URL: https://jsonschema.net/ (visited on 12/20/2022).

[12] Meike Klettke, Uta Störl, and Stefanie Scherzinger. “Schema extraction and structural outlier detection for JSON-based NoSQL data stores”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015). Publisher: Gesellschaft für Informatik eV.

[13] Liquid Technologies Limited. *Free Online JSON to JSON Schema Converter*. URL: https://www.liquid-technologies.com/online-json-to-schema-converter (visited on 11/14/2023).

[14] Minecraft Wiki. *Custom biome*. Minecraft Wiki. Sept. 13, 2023. URL: https://minecraft.wiki/w/Custom_biome (visited on 09/26/2023).

[15] Mojang Studios. *Minecraft JSON Biome Data*. Version Minecraft Java 1.20.1.

[16] open source initiative. *Opensource.org Licenses*. Sept. 5, 2023. URL: https://api.opensource.org/licenses/ (visited on 09/05/2023).

[17] Python Software Foundation. *JSON encoder and decoder*. Python documentation. URL: https://docs.python.org/3/library/json.html (visited on 10/26/2023).

[18] QuickType. *quicktype/quicktype*. original-date: 2017-07-13T00:22:50Z. May 23, 2023. URL: https://app.quicktype.io/#l=schema (visited on 05/23/2023).

[19] S. Broekhuis. *sbroekhuis/InteractiveSchemaInferrer*. GitHub. 2023. URL: https://github.com/sbroekhuis/InteractiveSchemaInferrer (visited on 10/23/2023).

[20] saasquatch. *GitHub - saasquatch/json-schema-inferrer: Java library for inferring JSON schema from sample JSONs*. URL: https://github.com/saasquatch/json-schema-inferrer (visited on 02/06/2023).

[21] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. "Inferring Versioned Schemas from NoSQL Databases and Its Applications". In: *Conceptual Modeling*. Ed. by Paul Johannesson et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 467–480. ISBN: 978-3-319-25264-3. DOI: 10.1007/978-3-319-25264-3_35.

[22] SurveyJS. *SurveyJS - JavaScript Libraries for Surveys and Forms*. URL: https://surveyjs.io/ (visited on 10/26/2023).

[23] Edvin Syse. *TornadoFX*. URL: https://tornadofx.io/ (visited on 10/26/2023).

[24] USGS. *GeoJSON Summary Format*. Sept. 5, 2023. URL: https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php (visited on 09/05/2023).

[25] YUI Library. *Douglas Crockford: The JSON Saga*. Aug. 29, 2011. URL: https://www.youtube.com/watch?v=-C-JoyNuQJs (visited on 03/13/2023).