MetaConfigurator: A Schema-Aware GUI Tool for Editing YAML/JSON Configuration Files

Santosh Kumar

santoshkuidev@gmail.com

Abstract. This paper presents MetaConfigurator, a tool designed to generate customizable graphical user interfaces (GUIs) for editing YAML and JSON configuration files based on a provided data schema. By leveraging the expressiveness of the JSON Schema (draft 2020-12), MetaConfigurator offers users a flexible and efficient way to edit configuration files while maintaining the speed and control of traditional code editors. The tool removes the need for bespoke GUIs tailored to specific schemas, streamlining workflows for developers and non-technical users alike. MetaConfigurator's key features include schema-based data retrieval, file modification, and schema editing. Although it currently supports JSON Schema 2020-12, potential improvements include extending support to other schema drafts, integrating code generation, and enhancing support for YAML and comment preservation. User feedback from a study indicates that MetaConfigurator effectively addresses practical tasks, suggesting its applicability in real-world scenarios. Further refinements such as desktop integration, interactive tutorials, and expanded format support are proposed to enhance the user experience.

1 Introduction

Literary arrangements to structure information, like JSON, XML and YAML, are frequently utilized for setup records or to structure estimation or on the other hand research information, since they can be perused and kept up with by people, as well as deserialized and utilized by PC programs. The organization of information designs can be characterized by purported constructions, which characterize the standards the information needs to adjust to. Given an outline, it very well may be approved whether a specific document affirms to that blueprint. For effortlessness, we will call all record examples utilizing such organizations *configuration files*. Contingent upon the space of such configuration files, they can be intricate and tedious to alter and keep up with. Tooling, like graphical UIs, can essentially decrease manual endeavors and help the client in altering the documents. Those graphical UIs (GUIs), in any case, require starting work to be created, as well as nonstop exertion in being kept up with and refreshed when the fundamental information blueprint changes. We tackle this issue by fostering a web application that naturally creates such helping GUIs, in view of the given information blueprint. Our methodology varies from other blueprint to-UI approaches in following:

The device joins the help of a GUI with the adaptability and speed of a code proofreader by giving both in one view. This code proofreader is a word processor that helps with normal elements, for example, grammar featuring, auto-consummation, and blunder featuring. The composition can be altered utilizing a similar device and sort of view. We utilize a similar method to produce a GUI utilizing the *meta schema* of the diagram language. The meta diagram is the composition that characterizes the construction language itself. In this manner, the client is given a comparable view for altering the pattern with respect to altering the configuration files. We support more perplexing mapping highlights, like circumstances and requirements.

In section 2, we talk about related work and existing diagram designs, as well as pattern to-gui draws near. In section **??**, we assess existing composition dialects to track down the most appropriate one for this work. Section 4 portrays the plan and presents the design of *MetaConfigurator*. Then, in section 5, we cover the execution of *MetaConfigurator*. This incorporates a definite portrayal of the diagram preprocessing steps, which are important to help a portion of the high level pattern highlights. To accumulate criticism and check whether the device would be able and will be utilized in reality, we directed a client study, which is depicted in section 6. We examine the ramifications of the outcomes in section 7. At long last, we finish up our work in section 8.

2 Related Work

This segment covers existing pattern dialects and existing ways to deal with create UIs from them.

As our exploration is of a commonsense sort, we likewise consider dim writing like details of constructions or sites.

2.1 Schema Languages

Schema languages are formal dialects that determine the design, requirements, and connections of information, for instance in a data set or organized information designs.

As this work is worried about creating a GUI in light of a composition, we really want to pick a reasonable diagram language. The accompanying areas portray existing composition dialects. We will analyze them in section **??** to figure out which is the most appropriate one for this work.

JSON schema JSON is a typical information trade design for trading information with web administrations, yet additionally for putting away reports in NoSQL data sets, like MongoDB [33]. In view of the notoriety of JSON, there is likewise an interest for a pattern language for JSON. One such language is JSON schema [1, 40]. Listing **??** shows an illustration of a JSON outline and listing 1.2 shows an illustration of a JSON report that adjusts to the mapping.

JSON pattern has advanced to being the true standard diagram language for JSON documents [12]. Patterns for some famous configuration file types exist. *JSON mapping store* [29] is a site that gives north of 600 JSON pattern records for different use cases. The upheld document types incorporate for instance Docker create or OpenAPI records.

We comment that JSON pattern and other outline dialects for JSON can likewise be applied to YAML as JSON and YAML reports are of a comparative construction (JSON is a subset of YAML). A few linguistic subtleties of YAML can, be that as it may, not be communicated with JSON diagram.

```json
{
  "$id": "https://example.com
  /person.schema.json",
  "$schema": "https://json-schema.org
  /draft/2020-12/schema",
  "title": "Person",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string",
      "description": "first name."
    },
    "lastName": {
      "type": "string",
      "description": "last name."
    },
    "age": {
      "description": "Age",
      "type": "integer",
      "minimum": 0
    }
  }
}
```

Listing 1.1: JSON schema example

```json
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 21
}
```

Listing 1.2: JSON example for the schema in listing 1.1

XSD and DTD For XML the two true standard construction dialects are Record Type Definition (DTD) [15] and XML Pattern Definition (XSD) [20]. XSD is the fresher and more expressive organization and in huge parts replaces and overrides the more restricted design DTD [14]. It is suggested by W3C as a diagram language for XML documents [20]. Numerous other construction dialects have been proposed and grown yet are generally obscure contrasted with XSD [31, 34].

Other schema languages We likewise think about the accompanying pattern dialects:

(a) Signal (Design, Bring together, Execute) [9] is an information approval and setup language, which can be utilized with different information designs, like JSON and YAML (it is a superset of both). It has a few use cases, particularly in setup and information approval.
(b) Apache Avro [3] is an open-source project that gives information serialization and information trade administrations for Apache Hadoop. It utilizes a JSON-based mapping language.
(c) JSON Type Definition (JTD) [17] is a mapping language for JSON records, which is fundamentally easier than JSON blueprint.
(d) Type Schema [28] is a diagram language for JSON reports, like JSON Type Definition yet utilizing an alternate punctuation.
(e) GraphQL diagram language [23] is a pattern language for GraphQL APIs.
(f) Convention Buffers [4] is a language for information serialization by Google.

We think about no graphical displaying dialects, for example, UML or trama center charts, as they are not text-based. In spite of the fact that they can be switched over completely to message based designs, their principal object is to demonstrate information designs and connections between them. We likewise think about no metaphysics dialects, like OWL or RDF Pattern, as they are not expected for information approval yet rather for information portrayal. Future work could explore in the event that such dialects are likewise helpful for our utilization case. At last, we think about no programming dialects as composition dialects. In fact, programming dialects can be utilized to characterize information designs and limitations, however they are not planned for this reason, and creating a GUI from them would very challenge.

2.2 Existing Approaches

Our work focuses on assisting users in creating and maintaining configuration files so that they are valid and adhere to a predefined schema.

There exist techniques to validate configuration files against a schema [10, 11, 13]. Usually, schema validation is done only internally, e.g., by web services or libraries. However, there exist also approaches that use the schema to assist the user in creating and maintaining configuration files. IDEs, such as Visual Studio

Code or IntelliJ IDEA, can validate configuration files against a schema and provide the user with error messages. Those IDEs also provide other features, such as auto-completion, syntax highlighting, and tooltips. However, they typically do not provide a graphical user interface (GUI) for editing the configuration files based on the schema.

Form generation Connected with our work are approaches that produce a GUI from a blueprint. This part covers structure generators, i.e., approaches that create a web structure from a mapping. Such structures can help the client in a huge number of ways, for example, by tooltips, auto-culmination (Figure 1) and dropdown menus (Figure 2). By innately sticking to the blueprint structure (generally speaking), altering information with such GUIs fundamentally diminishes arrangement botches brought about by the client. Clients who are not exceptionally acquainted with the arrangement construction benefit most from the GUI help, yet even experienced clients benefit from it.

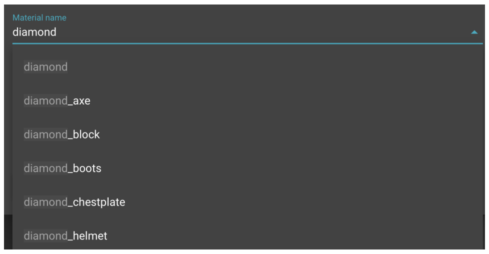


Fig. 1: Auto-Completion

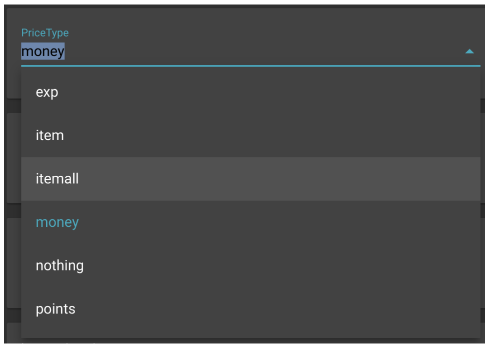


Fig. 2: Choice Selection

There exist different methodologies that produce web structures from a pattern, for various frontend systems, e.g., *React JSON Composition Form* [39], *An-*

gular Outline Form [26], *Vue Structure Generator* [25], *JSON Forms* [35], *JSON Editor* [27], and *JSON Form* [46]. Those approaches are undeniably founded on JSON pattern and create a structure that can be finished up by the client and the subsequent JSON report is approved against the diagram. On the off chance that the client enters invalid information, the structure shows a mistake message. The produced frames normally have a particular part for each kind of information, e.g. a text field for strings or a number field for numbers, like our methodology. Figure 3 shows an illustration of a produced structure utilizing JSON Structures.

Those strategies, nonetheless, just give the GUI to altering the information, however not a text-based supervisor. A text-based manager is helpful, particularly for experienced clients, who like to straightforwardly alter the information. Likewise, these procedures don't give a method for editting the actual diagram, yet just the information. The main impediment of all with the exception of the last two of the given methodologies is that they likewise require a "UI diagram" notwithstanding the JSON pattern, which is utilized to design the created structure. While these designs can be utilized to redo the produced structure, they likewise should be made and kept up with by the outline creator. Thusly, those approaches can't be utilized to produce a GUI for any erratic construction, yet manual exertion is expected to make the UI mapping.

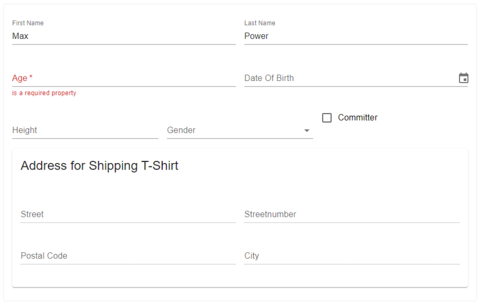
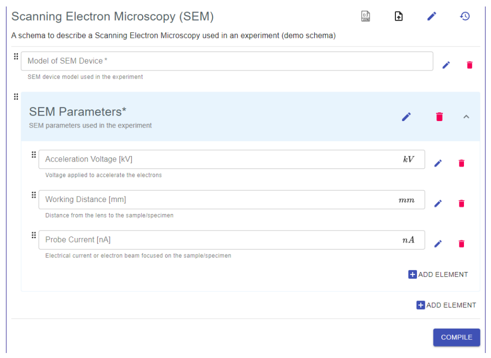

First Name
Max
Last Name
Power
Age *
is a required property
Date Of Birth
Height
Gender
☐ Committer
Address for Shipping T-Shirt
Street
Streetnumber
Postal Code
City

Fig. 3: JSON Forms, example for a generated form

Adamant [43] is a JSON-mapping based structure generator explicitly intended for logical information. It produces a GUI from a JSON mapping, permits altering and making JSON outline records furthermore, separates between a mapping alter mode and an information alter mode. It upholds a subset of JSON mapping, which is adequate for the overwhelming majority use cases. What's more, it upholds the extraction of units from the portrayal of a field, which is useful for logical information. Figure 4 shows a model in the construction alter mode.

Constraints of Determined are first, it really does just help a subset of JSON mapping, which is adequate for the majority use cases, yet not so much for any erratic pattern. Second, it doesn't give a text-based supervisor to neither the outline nor the information. At long last, it is explicitly intended for logical information, which makes it less reasonable for other use cases, particularly enormous and complex constructions.


Scanning Electron Microscopy (SEM)

A schema to describe a Scanning Electron Microscopy used in an experiment (demo schema)

Model of SEM Device *

SEM device model used in the experiment

SEM Parameters*

SEM parameters used in the experiment

Acceleration Voltage [kV] kV

Voltage applied to accelerate the electrons

Working Distance [mm] mm

Distance from the lens to the sample/specimen

Probe Current [nA] nA

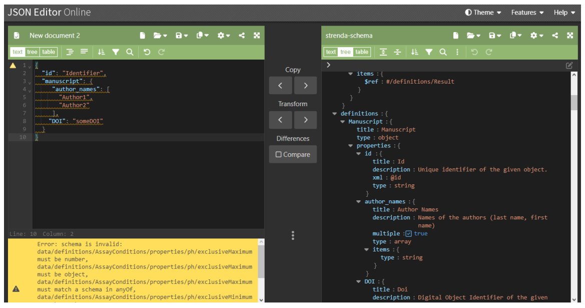
Electrical current or electron beam focused on the sample/specimen

Fig. 4: Adamant, example for a form in edit mode

Schema editors In *MetaConfigurator* we expect to give a GUI to both altering configuration files and altering the blueprint. For the last option, there exist a few purported diagram editors, which are instruments for making and altering constructions that are either text-based or graphical (or both).

JSON Supervisor Online [27] is an electronic proofreader for JSON compositions and JSON records. It isolates the proofreader into two sections, where one section can be utilized to alter the outline and the other part can be utilized to alter a JSON report, which is approved against the outline. The supervisor gives different elements, for example, language structure endlessly featuring of approval blunders (Figure 5). It gives a text-based or tree-based view for altering the JSON records. For straightforward articles that are not additionally settled, it likewise gives a table-based view (Figure 6). Notwithstanding, the elements of the manager are extremely restricted. For instance, it gives no help to the client, for example, tooltips or auto-fulfillment. For new records, it doesn't show the properties of the construction, so the client needs to know the pattern in advance.

There likewise exists an assortment of pattern editors that are paid programming, like *Altova XMLSpy* [8], *Liquid Studio* [32], *XML ValidatorBuddy* [7], *JSONBuddy* [2], *XMLBlueprint* [6], furthermore, *Oxygen XML Editor* [5]. Those are editors for XML or JSON pattern, for the most part with a blend of text-based and graphical perspectives. These instruments are not electronic and not open-source. Moreover, they don't zero in on altering a JSON report in view of a pattern, but instead just on altering the actual diagram.



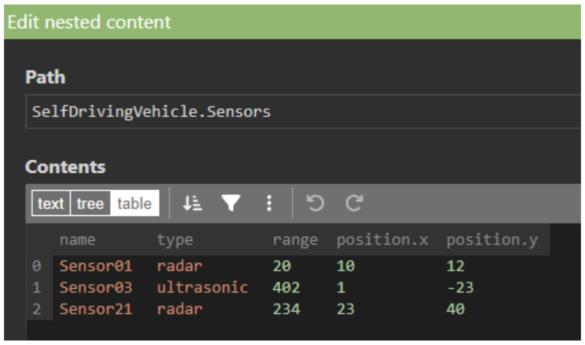
```
{
  "id": "Identifier",
  "manuscript": {
     "author_names": [
        "Author1",
        "Author2"
     ],
     "DOI": "someDOI"
  }
}
```

```
Line: 10  Column: 2
```

```
Error: schema is invalid:
data/definitions/AssayConditions/properties/ph/exclusiveMaximum
must be number,
data/definitions/AssayConditions/properties/ph/exclusiveMaximum
must be object,
data/definitions/AssayConditions/properties/ph/exclusiveMaximum
must match a schema in anyOf,
data/definitions/AssayConditions/properties/ph/exclusiveMinimum
```

```
    items : {
      $ref : #/definitions/Result
    }
  }
}
definitions : {
  Manuscript : {
    title : Manuscript
    type : object
    properties : {
      id : {
        title : Id
        description : Unique identifier of the given object.
        xml : @id
        type : string
      }
      author_names : {
        title : Author Names
        description : Names of the authors (last name, first
                      name)
        multiple : ✓ true
        type : array
        items : {
          type : string
        }
      }
      DOI : {
        title : Doi
        description : Digital Object Identifier of the given
```

Fig. 5: JSON Editor Online



```
SelfDrivingVehicle.Sensors
```

	name	type	range	position.x	position.y
0	Sensor01	radar	20	10	12
1	Sensor03	ultrasonic	402	1	-23
2	Sensor21	radar	234	23	40

Fig. 6: JSON Editor Online, table view

Schema visualization Generating a GUI from a schema is related to schema visualization, for which several techniques exist [18, 21, 38, 44]. However, the focus of schema visualization is on providing a static visual representation of the schema and not on providing a GUI for editing the schema. Thus, we do not consider schema visualization approaches in this work. However, future work can investigate how such techniques could be embedded in our approach.

3 Evaluation of Schema Languages

We evaluate the schema languages mentioned in section 2.1 to determine which is the most suitable one for this work.

3.1 Evaluation Criteria

Ideally, the schema language of *MetaConfigurator* is both popular and supported by numerous tools and libraries as well as expressive enough to express the features we need. We use the following criteria and metrics:

1. **Practical usage** — Ideally our approach uses a schema language that is already known by many developers. As indicator of the practical usage we

use the approximate search results on stackoverflow.com as metric. We acquire the results by querying the google search engine with the name of the schema language and “site:stackoverflow.com”, which limits the search results to stackoverflow.com. This metric might also correlate with the complexity of the schema language as a more complex to use schema language will likely lead to more questions asked on the site. Nevertheless, we assume that a significantly higher number of results indicates that a language is more known than others.

Additionally, we investigate how well the schema languages are supported by IDEs and code libraries:

(a) *Tool support* — We used the 10 most popular IDEs [16] and checked if the IDE supports the schema language either natively or by a plugin. Here, support means that either the IDE is capable of validating documents against a schema in the schema language or supports creating schema files, e.g., by using syntax highlighting for the schema language.

(b) *Library support* — As we implement a web-based tool, JavaScript or TypeScript based tools are helpful for our approach, e.g., so we can reuse a package for schema validation. We investigate the number of node modules existing that are related to the schema languages by querying the node module search on `www.npmjs.com` with the name of the schema language.

2. **Expressiveness** — We evaluate how expressive each of the schema languages is, i.e., what possible constructs the language is able to express. We define eight requirements on the language features that we consider helpful for our approach. The number of requirements a schema language fulfills is our metric that indicates how expressive the language is. Table 2 reports the results. The nine requirements are:

   (a) *Simple types* — This is fulfilled if the schema language provides the possibility to define simple data types, at least strings, numeric types, and a boolean type. This is a fundamental feature for our approach.

   (b) *Complex types* — This is fulfilled if the schema language provides the possibility to define complex data types, at least records and arrays. This is crucial feature for our approach as configuration files are often structured data rather than plain key-value pairs.

   (c) *Descriptions* — This is fulfilled if the schema language provides the possibility to add descriptions to fields. This is helpful in a schema-to-GUI approach as the description can be shown to the user, providing potential helpful information on how a field should be filled.

   (d) *Examples* — This is fulfilled if the schema language provides the possibility to add example values. This is helpful in our approach as the example values can serve as placeholders in the GUI editor.

   (e) *Default values* — This is fulfilled if the schema language provides the possibility to add default values which are assumed in an absence of a value. This helpful information can be displayed to the user or used as placeholder values.

(f) *Optional values* — This is fulfilled if the schema language provides the possibility to declare values as optional or required. Often it is not necessary to provide all values in a configuration file, so it is helpful to mark fields as required or optional in the GUI editor.

(g) *Constraints* — This is fulfilled if the schema language provides the possibility to constrain values of fields, e.g., maximum length of strings. To be exact, for this evaluation we require that at least two of the following constraints can be expressed by the schema language:
- The length of strings can be limited.
- The range of numeric types can be limited, e.g., to only positive values.
- The valid values of a field can be restricted to a finite amount of values (enumeration).
- The format of a string field can be constrained to a certain pattern.

This is a helpful feature for our approach as often not all possible values are valid for specific fields in configuration files.

(h) *Conditions* — This is fulfilled if the schema language provides the possibility to define conditional dependencies between fields. This is an advanced feature that is helpful because it allows to express that a particular field must be given only if another field has a specific value.

(i) *References* — This is fulfilled if the schema language provides the possibility to define reusable sub-schemas that can be referenced in other parts of the schema. This is often useful in practice to reuse common data structures.

Table 1
Evaluation of different schema languages

Schema language	**# Search results**	**IDE support**	**# Node packages**	**Expressiveness**
JSON schema	245,000	8 / 10	4,536	9 / 9
XSD	151,000	8 / 10	116	8 / 9
DTD	69,700	9 / 10	34	6 / 9
CUE	10,500	4 / 10	97	8 / 9
Avro	20,000	8 / 10	211	5 / 9
JTD	109	0 / 10	17	5 / 9
TypeSchema	8,450	0 / 10	5	8 / 9
Protobuf	44,800	9 / 10	1,210	4 / 9
GraphQL schema	31,000	7 / 10	1,509	6 / 9

3.2 Evaluation results

Tables 1 and 2 show the results of our evaluation. We come to the conclusion that JSON schema is sufficiently popular and expressive that we choose to use it

Table 2
Comparison of expressiveness of different schema languages (Part 1)

Schema language	**Simple types**	**Complex types**	**Descriptions**	**Example values**	**Default values**
JSON schema	✓	✓	✓	✓	✓
XSD	✓	✓	✓	x	✓
DTD	✓	✓	x	x	✓
CUE	✓	✓	✓	✓	x
Avro	✓	✓	x	x	✓
JTD	✓	✓	x	x	x
TypeSchema	✓	✓	✓	x	✓
Protobuf	✓	✓	x	x	x
GraphQL schema	✓	✓	✓	x	✓

as the schema language for our approach. The other schema languages are either less expressive or less popular. This result is in line with the work of Baazizi et al. [12], who also found over 80,000 JSON schema files on GitHub, and with their claim that JSON schema is the de-facto standard for JSON schema languages.

4 Design

4.1 User Interface

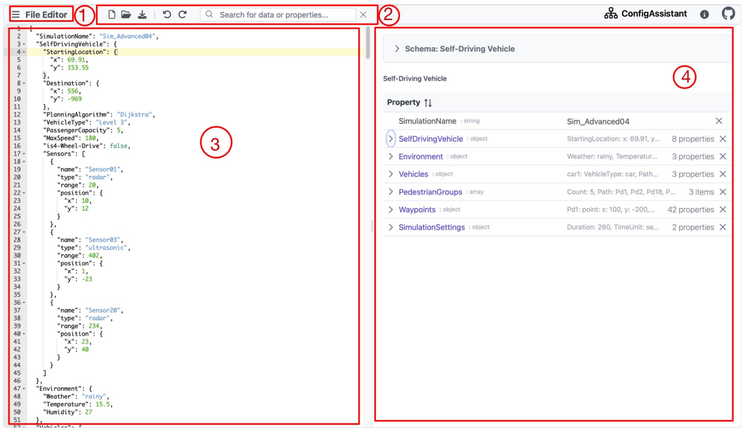
The design of *MetaConfigurator* is inspired by another tool [36] by one of the authors, which is a GUI program that assists users in editing configuration files of BossShopPro [37]. That tool provides the user a code panel for editing configuration files of that domain in a text editor, as well as a GUI panel, where the user can edit their configuration file using GUI components. *MetaConfigurator* differs from that tool by being generic, instead of being bound to a certain domain, by having a much more expressive schema language, a schema editor, and many other features that improve the user experience, such as a search functionality.

Before we dive into the architecture and detailed design of *MetaConfigurator*, this section provides an overview from the view of the user.

The UI has three particular perspectives:

1. Document proofreader (figure 7): In this view, the client can change their configuration file, in light of a composition.
2. Outline proofreader (figure 8): In this view, the client can change their composition.
3. Settings (figure 13 in supplement): In this view, the client can change boundaries of the apparatus.

The UI is separated into two fundamental boards: the *Code panel* (on the left) and the *GUI panel* (on the right). In the *Code panel* the client can change

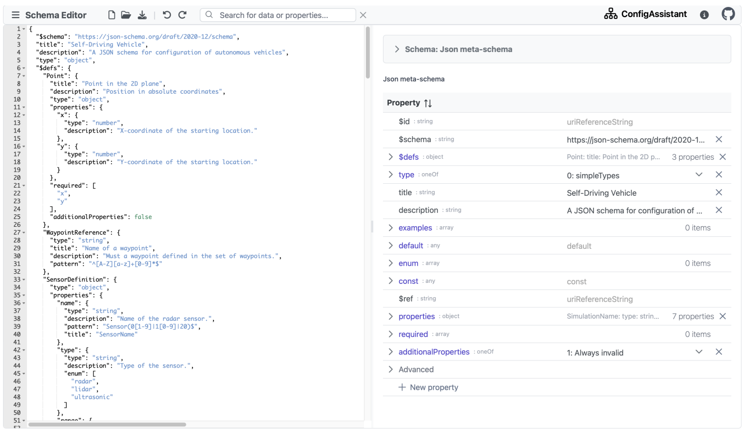

File Editor
1
Search for data or properties...
2
ConfigAssistant
3
Schema: Self-Driving Vehicle
4

```
{
  "SimulationName": "Sim_Advanced04",
  "SelfDrivingVehicle": {
    "StartingLocation": {
      "x": 69.91,
      "y": 153.55
    },
    "Destination": {
      "x": 556,
      "y": -969
    },
    "PlanningAlgorithm": "Dijkstra",
    "VehicleType": "Level 3",
    "PassengerCapacity": 5,
    "MaxSpeed": 180,
    "is4-Wheel-Drive": false,
    "Sensors": [
      {
        "name": "Sensor01",
        "type": "radar",
        "range": 20,
        "position": {
          "x": 10,
          "y": 12
        }
      },
      {
        "name": "Sensor03",
        "type": "ultrasonic",
        "range": 402,
        "position": {
          "x": 1,
          "y": -23
        }
      },
      {
        "name": "Sensor20",
        "type": "radar",
        "range": 234,
        "position": {
          "x": 23,
          "y": 40
        }
      }
    ]
  },
  "Environment": {
    "Weather": "rainy",
    "Temperature": 15.5,
    "Humidity": 27
  },
```

Self-Driving Vehicle

Property ↑↓		
SimulationName : string	Sim_Advanced04	
SelfDrivingVehicle : object	StartingLocation: x: 69.91, y...	8 properties
Environment : object	Weather: rainy, Temperatur...	3 properties
Vehicles : object	car1: VehicleType: car, Path...	3 properties
PedestrianGroups : array	Count: 5, Path: Pd1, Pd2, Pd18, P...	3 items
Waypoints : object	Pd1: point: x: 100, y: -200,...	42 properties
SimulationSettings : object	Duration: 260, TimeUnit: se...	2 properties

Fig. 7: UI of file editor view. Different components highlighted in red: 1) button to switch to other view (e.g. to Schema Editor view), 2) Toolbar with various functionality, 3) Code panel, 4) GUI panel



```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Self-Driving Vehicle",
  "description": "A JSON schema for configuration of autonomous vehicles",
  "type": "object",
  "$defs": {
    "Point": {
      "title": "Point in the 2D plane",
      "description": "Position in absolute coordinates",
      "type": "object",
      "properties": {
        "x": {
          "type": "number",
          "description": "X-coordinate of the starting location."
        },
        "y": {
          "type": "number",
          "description": "Y-coordinate of the starting location."
        }
      },
      "required": [
        "x",
        "y"
      ],
      "additionalProperties": false
    },
    "WaypointReference": {
      "type": "string",
      "title": "Name of a waypoint",
      "description": "Must a waypoint defined in the set of waypoints.",
      "pattern": "^[A-Z][a-z]+[0-9]*$"
    },
    "SensorDefinition": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Name of the radar sensor.",
          "pattern": "Sensor(0[1-9]|1[0-9]|20)$",
          "title": "SensorName"
        },
        "type": {
          "type": "string",
          "description": "Type of the sensor.",
          "enum": [
            "radar",
            "lidar",
            "ultrasonic"
          ]
        },
```

Json meta-schema

Property ↑↓		
$id : string	uriReferenceString	
$schema : string	https://json-schema.org/draft/2020-1...	
$defs : object	Point: title: Point in the 2D p...	3 properties
type : oneOf	0: simpleTypes	
title : string	Self-Driving Vehicle	
description : string	A JSON schema for configuration of ...	
examples : array		0 items
default : any	default	
enum : array		0 items
const : any	const	
$ref : string	uriReferenceString	
properties : object	SimulationName: type: strin...	7 properties
required : array		0 items
additionalProperties : oneOf	1: Always invalid	
Advanced		
+ New property		

Fig. 8: UI of schema editor view

their information the hard way, as in an ordinary code supervisor. Highlights, for example, punctuation featuring and composition approval, help the client. In the *GUI panel*, the client can change their information with the assistance of a GUI. The GUI depends on the blueprint which the client gives (to a greater degree toward that in the following passages). For instance, for enum properties, the client will get a dropdown menu with the various choices to browse and for boolean properties the client will get a checkbox. Different elements, for example, tooltips that show the depiction, and imperatives of a property, further help the client.

This plan consolidates the advantages of both a code manager with the advantages of a GUI. A code manager is effective for some errands and more appropriate for clients with a specialized comprehension of the information structure, while a GUI empowers clients without profound specialized understanding to work with the information and furthermore helps master clients.

As a pattern is a configuration file itself, it very well may be treated thusly and the device can offer help in like manner. Note that at whatever point the client alters a configuration file utilizing the instrument, they do so utilizing some basic pattern. Indeed, even the instrument settings should be visible as a configuration file, for which the hidden outline is a settings construction.

Table 3 delineates how for the various perspectives, record information and outline being utilized by the instrument contrast.

Table 3
File data and schema for the different views

View	**Effective File Data**	**Effective Schema**
File editor	User data	User schema
Schema editor	User schema	JSON Schema Meta Schema
Settings	Settings data	Settings schema

4.2 Intended Workflow

1. Upon initial access of *MetaConfigurator*, a dialog is displayed, where users can select their desired schema.
2. After selecting a schema, the user will find that a GUI is automatically generated on the right-hand side of the file editor, tailored to the selected schema.
3. Through the GUI panel, users are assisted in creating or modifying configuration files.
4. If a user wishes to modify the selected schema, such as adding new properties, they can do so through the schema editor. Changes can be made using either the GUI panel or the code panel, and these modifications will automatically reflect in the file editor.

4.3 Architecture

This section describes our main architectural design decisions. Those will not be relevant or visible for the user of the tool but are important for understanding our implementation. The aim of those design decisions is to ensure modularity and maintainability of the tool.

The core of *MetaConfigurator* is a single source of truth data store that contains the current user configuration data (as a JavaScript Object). With this data store, we can bidirectionally connect what we call "editor panels". An editor panel is a modular component that the user can access to modify the data indirectly. It might be implemented as a code editor, a graphical user interface, or any other way in which the data can be presented to the user. All editor panels are independent and only have access in the data store but not to each other. Every editor panel subscribes to the changes of the data store, so it can be updated accordingly whenever the data in the store is changed. Additionally, every panel has the capability of updating the data store themselves, which is done when the user modifies the data in the editor panel. The following artificial example use cases illustrate the capabilities of this architecture:

- Format converter: one panel shows the data in a code editor in JSON format, and a second panel shows the data in a code editor in YAML format. Any semantic data change on one panel will cause the same semantic change in the other panel.
- Split-Screen Editor: one panel shows the data in a code editor, and a second panel shows the data in a GUI. This way the user can have the efficiency of a text editor, but also the assistance of a GUI at the same time. Any semantic data change on one panel will be forwarded to the other panel.
- The Split-Screen Editor could be implemented for different data formats, such as YAML, JSON, and XML. The architecture allows any data format as long as there exists a mapping from this data format to a JavaScript Object and back.

In practice, we implement only one code editor panel, as well as one GUI editor panel. The architecture, however, would be flexible enough to allow replacing any of these panels or adding new ones, since they are decoupled from each other and only communicate with the single source of truth data store.

Single Source of Truth Data Store This is the core of the tool. The panels can subscribe to this store to receive updates whenever data is changed. Also, panels can trigger changes in the data in the store. Besides the current configuration data, the store also stores the path of the currently selected data entry and the schema that is currently being used.

Code Panel For the code board, we install a code proofreader that as of now upholds sentence structure featuring and other helpful highlights. We empower approval of whether the text is very much framed by the JSON/YAML/XML standard and add mapping approval. The board buys into the information store. Whenever the arrangement information is changed in the store, the board will take the new setup information JavaScript Article, serialize it into the given information design, and supplant the text in the code proofreader with the new serialized information. The activity of supplanting the text in the code manager will make designing and remarks be lost, which we acknowledge. Later on, a few components can be applied to keep away from the deficiency of designing or remarks (see segment 7.2).

At the point when the client alters the message in the code manager, the message is deserialized into a JavaScript Item and shipped off the information store, which then refreshes the setup information item and tells any remaining bought in boards of the change.

To empower correspondence with the store, for any information design that the device ought to help, we want a capability to change over a JavaScript object to a string in the information design and a capability to parse a string in that information design as a JavaScript object.

To make it conceivable to feature specific lines in the manager as wrong (construction infringement) or leap to specific lines (for example at the point when the client chooses a property in the GUI supervisor, we need to leap to a similar property in the word processor), we want a capability `determineRow(editorContent, dataPath)` that can decide the relating proofreader line, in light of the design text and a given information way.

The reverse way around, when the client puts their cursor inside the word processor, we need to decide the way of the component that the cursor is as of now at. This requires a capability `determinePath(editorContent, cursorPosition)` which returns an information way founded on the design text and a given cursor position.

GUI Assistance Panel The GUI assistance panel directly works with the given schema and provides the user with corresponding GUI elements, such as a checkbox for a boolean data structure or a text field for a string data structure. Additional GUI elements, such as tooltips (showing the description of a data field) are used to support the easier. The GUI elements are constructed in the following manner: a schema is seen as a hierarchical tree of data field definitions and their corresponding constraints. A data field can either be simple (string, boolean, number, integer) or complex (array or object with children). Every schema has a root data field. The GUI element for this root data field is constructed. When constructing the GUI element for a complex data field, all GUI elements of the child data fields are constructed too, in a recursive manner. This way, the whole schema tree is traversed and GUI elements for all entries are constructed. To avoid overwhelming the user with too many GUI elements, the ones with child elements can be expanded or collapsed by the user and only a limited amount of them is expanded by default. By design, each of these constructed GUI elements is mapped to their corresponding data field (in other words: to a path in the data structure). The initial values of all GUI elements are taken from the data in the store, by accessing the data at the given paths. Whenever the values in a GUI element are adjusted by the user, the data in the store will be updated with the new values.

5 Implementation

This section contains the implementation details of *MetaConfigurator*. We first describe the implementation and features of the two main components of *MetaConfigurator*, the code editor and the GUI editor. Then, we explain the schema preprocessing steps that are required to generate the GUI editor. Finally, we describe how we developed a new meta schema for JSON schema that is more suitable for our tool.

5.1 Technologies

We use Vue.js [45] as the UI framework for our tool, combined with the component library PrimeVue [30].

5.2 Code Panel

The code editor is a GUI panel designed for editing the configuration files. For this project, we use the *Ace Editor* [24] library to embed an interactive code editor into our user interface. It provides useful features for our approach, such as syntax highlighting and code folding. To make our code editor more user-friendly, we implemented several features, which are described in the following.

Schema Validation To provide the user feedback on whether their data is valid according to the provided schema, we perform schema validation. We make use of the *Ajv JSON schema validator* [19] library, which supports the newest JSON schema draft 2020–12.

If schema violations are found, the corresponding user data lines in the code editor will be marked with a red error hint, which also describes the violation.

Linkage of text with the data model As described in section 4.3, to map a cursor position in the text editor to a path in the data model we need to implement the function `determinePath(editorContent, cursorPosition)` and to map a path in the data model to a text row in the editor, we need to implement the function `determineRow(editorContent, dataPath)`.

For *JSON*, the functions have been implemented using a *Concrete Syntax Tree (CST)*. The text content is parsed as a CST. Then this tree is traversed recursively. Every tree node has a range property, describing the start and end index of the text belonging to the node. To determine the corresponding path for a cursor position, the cursor position is translated to a character index `targetCharacter` within the text. Then the CST is traversed and for all nodes `currentNode` of type array or object for which `targetCharacter` $\in$ `currentNode.range`, the child nodes are checked, and the key of the node (or index for array elements) is appended to the result path. This way, the corresponding path is built up.

To determine the cursor position for a given path, we reverse this algorithm: the CST is traversed until the node is found whose path matches the target path. Then `currentNode.range.start` is returned as the result index, which is then translated into a cursor position (row and column).

For *YAML*, this linkage is not yet implemented and will be part of further work.

Editor Operations The code editor has more functionalities, such as the possibility to open a file by drag and drop into the editor, undo/redo operations, and the possibility to change the font size.

5.3 GUI Panel

The GUI supervisor is a part that permits the client to alter the design information in a GUI, which is created in light of the pattern of the design information. It is organized in a table-like way, where each line addresses a key-esteem sets of the setup information. Cluster components are addressed in much the same way, where the file of the exhibit component is the key and the worth is the exhibit component itself. Figure 7 shows the GUI manager part with a model pattern and setup information.

To permit this portrayal of the pattern, we do some preprocessing of the blueprint, which is portrayed in section 5.4. To help the client in altering the design information, the GUI manager offers a bunch of elements, which are depicted in the accompanying.

Traversal of the Data Tree As is normally done, only the chief level of the data tree is shown. The client can expand the data tree by tapping on the bolt near the key of a thing or display. This will show the sub-properties of the thing or the parts of the group. We limit the significance of the data tree to a configurable worth, to thwart the GUI manager ending up being unnecessarily overwhelming. Regardless, the client can similarly tap on the property name or group document to *zoom in* on that part. This will show the sub-properties of that part at the undeniable level, like that property were the foundation of the data tree. The breadcrumb at the top allows the client to see what direction the GUI editor at this point shows and to investigate back to the upper levels of the tree.

GUI implementations for JSON schema features Table 4 shows how the different JSON diagram highlights are carried out in our GUI.

Table 4
Corresponding GUI implementations for JSON schema features

JSON schema type / keywords	**GUI implementation**
string	Text field.
number	Text field which allows only floating point numbers and has buttons to increment and decrement.
integer	Text field which allows only integer numbers and has buttons to increment and decrement.
boolean	Checkbox.
object	Expandable list of child columns in the properties table (see figure 9).
array	Similar as for object. Also has a button to add new items (see figure 10).
enum	Dropdown menu.
const	Dropdown menu with just one entry.
required	Red asterisk left of the property name.
deprecated	Strikethrough styling for property name.
anyOf	Multiselect menu to choose sub-schemas. Based on the selected sub-schemas, corresponding properties will be shown as children in the table.
oneOf	Dropdown menu to choose one sub-schema. Based on the selected sub-schema, corresponding properties will be shown as children in the table.

Remove Data The client can erase properties or exhibit components from the information by tapping on the × button close to the alter field. This button is possibly shown on the off chance that the property isn't needed and there exists information.

Schema Information Tooltip At the point when the client floats over the property key or exhibit file, an overlay is shown (figure 11), which contains all data from the mapping about that property. We physically carried out an age of a literary depiction for every one of the JSON mapping watchwords. Beginning with the title and portrayal of the property, the overlay then, at that point, shows imperatives, (for example, the number should be more prominent than 0) and at the base, it likewise shows mapping infringement, in the event that there are any. This element assists the client with grasping the imperatives and the importance of a property.

Highlighting Schema Validation Errors When the configuration data does not comply with the schema, the corresponding elements are underlined in red and highlighted with a red error icon. This way, the user knows what parts of

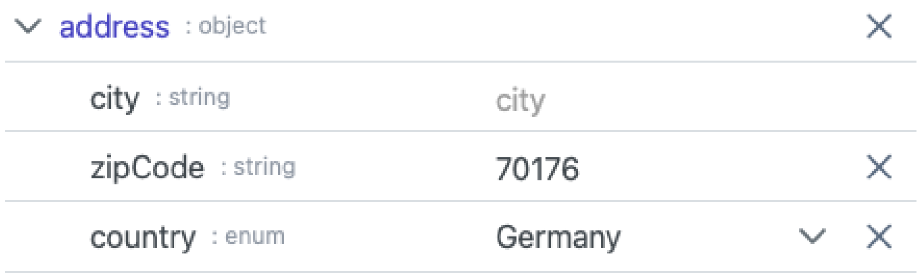


Fig. 9: GUI implementation for object properties

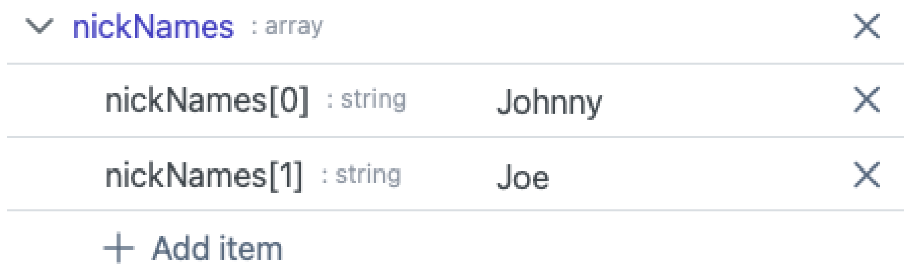


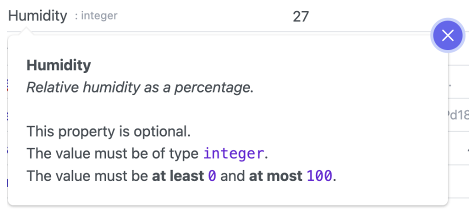
Fig. 10: GUI implementation for array properties

the data are invalid and what the error is. Additionally, the schema information tooltip lists all schema violations, as shown in figure 12.

5.4 Schema preprocessing

To address the blueprint in the GUI supervisor, it is important to preprocess the composition concerning some JSON pattern watchwords it isn't quickly clear how to address them in a GUI. For instance, the `type` catchphrase can have numerous qualities, which address a sort association. There is no undeniable GUI part that addresses this case. We separate between three different ways of preprocessing: A one-time preprocessing step while stacking the blueprint, an interior preprocessing that occurs at each layer of the composition tree, also, computing a viable outline that happens each time the setup information changes. It is essential to take note of that every one of these preprocessing steps are just utilized for creating the GUI manager. They won't influence the pattern document itself that is stacked into the outline supervisor. The accompanying areas depict the preprocessing steps exhaustively and how we settle cases like the one recently referenced.

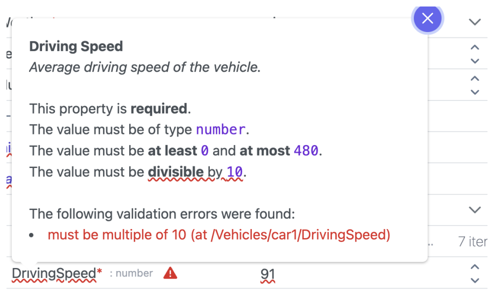
One-time Preprocessing Step At the point when the construction is stacked, we play out a one-time preprocessing step. This step just cycles the entire pattern


Humidity : integer 27

Humidity
Relative humidity as a percentage.

This property is optional.
The value must be of type `integer`.
The value must be **at least** `0` and **at most** `100`.

Fig. 11: Tooltip


Driving Speed
Average driving speed of the vehicle.

This property is **required**.
The value must be of type `number`.
The value must be **at least** `0` and **at most** `480`.
The value must be **divisible** by `10`.

The following validation errors were found:
- must be multiple of 10 (at /Vehicles/car1/DrivingSpeed)

DrivingSpeed* : number 91

Fig. 12: Tooltip with a schema violation

once and doesn't rely upon the design information. We play out no tedious activities in this step, so the client doesn't need to trust that the GUI will stack. The accompanying three stages are acted in this preprocessing step:

1. Title prompting: On the off chance that a property doesn't have the watchword `title`, we infuse the property name as `title`. The `title` can be then utilized in different spots in the GUI, for example, the tooltip as a placeholder in the text field.
2. Handling `enum` and `const`: The `const` catchphrase is utilized to limit the worth of a property to a solitary worth. It is semantically comparable to the `enum` watchword with a solitary worth. In this manner, we convert any utilization of `const` to `enum` with a solitary component, which permits us to overlook the `const` watchword in different tasks.
3. Deducing sorts of enums: If the `enum` watchword is utilized, however not the `type` catchphrase, we induce the kind of the property from the components of the `enum` exhibit. This is helpful for the GUI proofreader, as it permits us to show the right kind data in the tooltip.

Lazy preprocessing The accompanying preprocessing steps occur at each layer of the diagram tree sluggishly, just when the client extends the relating property in the GUI supervisor. Apathy of the preprocessing is expected as blueprints can have roundabout references, which would, somehow or another, lead to boundless circles. In the accompanying, we depict the preprocessing steps exhaustively.

Resolving references JSON schema uses the `$ref` keyword to reference other schemas. This can either be references to schemas in the same file (using the `$defs` keyword), references to other local files, or references to schemas at a URL on the web. We currently only support references to schemas in the same file. These are lazily resolved as the first preprocessing step. listing 1.3 shows an example schema, listing 1.4 shows the equivalent example after this first preprocessing step.

```
{
  "title": "NonEmptyString",
  "$ref": "#/$defs/nonEmptyString",
  "$defs": {
    "nonEmptyString": {
      "type": "string",
      "minLength": 1
     }
  }
}
```

Listing 1.3: Simple JSON schema before reference resolving

```
{
  "allOf": [
   {
     "title": "NonEmptyString"
   },
   {
     "type": "string",
     "minLength": 1
   }
  ],
  "$defs": {
    "nonEmptyString": {
      "type": "string",
      "minLength": 1
     }
  }
}
```

Listing 1.4: Simple JSON schema after reference resolving

```
{
  "title": "NonEmptyString",
  "type": "string",
  "minLength": 1,
  "$defs": {
    "nonEmptyString": {
      "type": "string",
      "minLength": 1
    }
  }
}
```

Listing 1.5: Simple JSON schema after allOf resolving

Resolving allOfs The `allOf` keyword in JSON schema specifies that all the schemas in the given array must be valid. To simplify any other operation on the schema, we aim to merge the schemas in the `allOf` array into one equivalent schema. As the first step, we do a recursive step by preprocessing all the schemas of the `allOf` array. Then, we use the *mergeAllOfs* [22] library to merge all the sub-schemas. Listing 1.5 shows the previous example schema after this step. It is important to note that this library only supports a few keywords of JSON schema, most notably the `properties` and `items` keywords. Hence, the *MetaConfigurator* has only limited support for `allOf` and any other keywords for which we use this library in the preprocessing.

Converting types to oneOf In JSON schema, a property can have multiple types, such as shown in listing 1.6. A semantically equivalent schema can be generated by the use of `oneOf`, where each sub-schema contains exactly one of the types, as shown in listing 1.7. If a schema defines more than one type, we convert the types to `oneOf`. As `oneOf`s are represented as a dropdown menu in the GUI editor, we now have a way to represent multiple types in the GUI. For schemas that already contain `oneOf`, every type is *multiplied* by every existing `oneOf` sub-schema. For two types and three `oneOf` sub-schemas, this will result in a new `oneOf` with six options. An exception is when a type can not be merged with a `oneOf` sub-schema (e.g., the `type` is "boolean" and the `oneOf` sub-schema has type "string"). In that case, the incompatible pair is dismissed.

```
{
  "type": ["object", "boolean", "string"]
}
```

Listing 1.6: Simple JSON schema with three possible types

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

Listing 1.7: Simple JSON schema after conversion of types to oneOf

Removing incompatible oneOfs and anyOfs A schema may have `oneOf` or `anyOf` options that are not compatible with the schema of the property (e.g., sub-schemas that can never be fulfilled in combination with the property schema). Listing 1.8 provides an example of a schema with an incompatible `oneOf` option. For every `oneOf` and `anyOf` sub-schema, we check whether it can be merged with the schema of the property. The options which are not compatible are removed (see listing 1.9).

```
{
  "type": "object",
  "oneOf": [
    {
      "type": "object"
    },
    {
      "type": "boolean"
    }
  ]
}
```

Listing 1.8: Simple JSON schema with incompatible oneOf option

Merging singular oneOfs and anyOfs Because of the previous pre-processing step, it can happen that for some `oneOf`s or `anyOf`s, there remains only one compatible sub-schema left (see listing 1.9). If this is the case, the use `oneOf`/`anyOf` is redundant, as that singular sub-schema must be chosen implicitly. Therefore,

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

Listing 1.9: Simple JSON schema with the incompatible oneOf option removed

if there exists only one singular choice for `oneOf/anyOf`, we merge its sub-schema into the property schema and remove the use of `oneOf`/anyOf (see listing 1.10).

```
{
  "type": "object"
}
```

Listing 1.10: Simple JSON schema with singular oneOf merged into property schema

Attempting to merge oneOfs into anyOfs Schemas can use both `anyOf` and `oneOf` at the same time. Especially after converting type unions to `oneOf`, it happens that a schema has `oneOf` options (typically for types) and simultaneously anyOf options. The user will then have to select both a `oneOf` sub-schema, as well as an `anyOf` sub-schema in the GUI. We observed a special scenario in the JSON meta schema, where the `oneOf` selection was always implicitly given by the `anyOf` selection. For every single `anyOf` sub-schema, only one `oneOf` sub-schema was compatible. In that scenario, we can merge the `oneOf`s into the `anyOf`s: for every `anyOf` sub-schema, we merge the one compatible `oneOf` sub-schema into it. This is precisely what this pre-processing step does: if possible, the `oneOf` sub-schemas are merged into the `anyOf` sub-schemas, and the `oneOf` property is removed from the schema.

Preprocessing oneOfs and anyOfs For all remaining `oneOf` and `anyOf` sub-schemas, the internal pre-processing steps are executed.

Calculating an effective schema This third preprocessing step is determined each time the information changes. The JSON mapping watchwords `if`, `then`, and `else` give a method for remembering conditions for the JSON pattern. In the event that the blueprint in the `if` field is legitimate, additionally the composition in the `then` field should be substantial, in any case, the pattern in the `else` field should be substantial. This makes the blueprint information subordinate. To show the right properties, we assess the information and ward on legitimacy or not, we either utilize the `then` or the `else` mapping. We comparatively handle the `dependentRequired` and the `dependentSchemas` catchphrases. For blueprints with no of those catchphrases, this step is minor as the composition isn't altered in any capacity.

```
{
  "properties": {
    "mode": {
        "enum": ["manual", "automatic"]
    }
  },
  "if": {
    "properties": {
      "mode": {
        "const": "manual"
      }
    },
    "required": ["mode"]
  },
  "then": {
    "properties": {
        "manualValue": {
            "type": "number"
        }
    }
  }
}
```

Listing 1.11: Data dependent schema. If the field `mode` is set to "manual" in the data, users will expect that the GUI shows the `manualValue` property

Listing 1.11 shows an example of a data-dependent schema and listing 1.12 shows the effective schema when the value for `mode` is "manual".

5.5 Developing a Custom Meta Schema

The schema editor view has the same structure as the file editor view, as discussed in previous sections. The only difference is that the schema used for generating the GUI panel is not the schema file provided by the user but the JSON schema meta schema, i.e., the schema that defines the structure of valid JSON schema files. However, applying our generic approach on the official JSON schema meta schema [1] does not result in a user-friendly editor for creating and modifying schema files. In this section, we discuss the reasons for that and how we developed a new meta schema that circumvents the problems of the official meta schema.

```
{
  "properties": {
    "mode": {
        "enum": ["manual", "automatic"]
    },
    "manualValue": {
        "type": "number"
    }
  }
}
```

Listing 1.12: Effective schema when the value for `mode` is “manual”

Missing descriptions With the `description` keyword, schema authors can give descriptions to any elements of their schema. This can help the user of a schema in many ways, for example, the author can specify the unit of a numeric field or give other additional information. The JSON schema meta schema does not provide any descriptions. Users, especially those without prior knowledge of JSON schema, might not understand the meaning of the fields of JSON schema. Thus, we insert descriptions from the JSON schema specification [1] into our modified meta-schema.

External references *MetaConfigurator* does not support references to external schemas yet, i.e., references inside the schema to a schema at a specific URL. Also, *MetaConfigurator* does not support the `$vocabulary` keyword. Both features are used in the JSON schema meta schema, as it is distributed over multiple schema files. To circumvent that problem, we put all schemas in one schema file into the `$defs` object and replace the external references with local references.

Use of dynamic anchors and references The JSON schema meta schema uses dynamic references and dynamic anchors. The difference between those keywords in comparison to the `$ref` keyword is that they provide a way to dynamically extend the JSON meta schema at runtime. For example, one could combine the JSON schema meta schema with an extension that defines how fields should be serialized in XML. We do not support dynamic references and anchors yet. We replaced all of them with “non-dynamic” references using the `$ref` keyword.

Allowing each field in each context The JSON schema meta schema allows each field in each context. For example, if the `type` keyword is used and set to `string`, then the `properties` keyword is allowed, even though it does not make sense in that context. According to the specification, any validator should ignore the fields that do not make sense in the current context. Consequently, if the user gets shown all fields that are allowed by the meta schema, they get overwhelmed, although many fields do not make sense in a given context. This is also feedback we got from our user study.

```
{
  "if": {
    "$ref": "#/$defs/hasTypeArray"
  },
  "then": {
    "$ref": "#/$defs/arrayProperty"
  }
}
```

Listing 1.13: If condition for array properties. The `hasTypeArray` is valid if the current property is of type array. The `arrayProperty` schema defines the properties of an array.

Thus, we added `if` conditions to each field to only show them when they make sense in the current context. Listing 1.13 shows an example of such an `if` condition. The relevant properties for arrays are only shown when the current property is of type array.

To even more reduce the amount of fields shown to the user, we also introduce a custom keyword for our own meta schema. The keyword `advanced` is a boolean field that is set to `false` by default. It is wrapped in an `metaConfigurator` object, which is ignored by any validator as it is not part of the JSON schema specification. We use this wrapper to prevent any other schema extensions from colliding with our keyword. When set to `true`, the field is not shown by default, but only when the user expands an *advanced* section that we added to the GUI. We put all fields that are not required for the basic usage of the schema into the advanced section. For this, we oriented ourselves on the work of Baazizi et al. [12], which analyzed the usage of JSON schema keywords in 82,000 JSON schemas.

6 User Study

We conduct a qualitative user study with five participants. During each interview, we introduce *MetaConfigurator*, give the participant tasks to execute using the tool, and finish the session with open-ended questions. We observe how the participants work with the tool and which difficulties they have when executing the tasks. Additionally, we ask them for feedback and improvement suggestions. We also conduct a survey to gather demographic information about the participants of the user study.

6.1 Research Questions

We address the following research questions with the user study:

1. **RQ1:** Which aspects of the tool can be improved?
2. **RQ2:** Are users able to perform the following types of tasks using the tool:
   - **RQ2.1** Retrieve information from configuration files in the context of a given schema
   - **RQ2.2** Modify configuration files within the constraints of a given schema
   - **RQ2.3** Modify a schema file
3. **RQ3:** Would people use the tool in practice?

6.2 Methodology

Our user study is qualitative and consists of five interviews. We do not perform any major quantitative analysis of the results as the sample size is too small to draw any statistically significant conclusions.

Potential Users We look for potential users to conduct the interviews. We consider the following groups of people as potential users:

- Professors and students who are interested in our application.
- People who frequently use configuration files and schema languages

In table 6, the professions of the participants are listed.

Interview Questions We created a JSON schema and configuration file for a made-up self-driving car simulation software. All interview questions are about working with these files. By using a made-up example, we avoid that the participants are biased by their domain knowledge. We note, however, that besides the user study, we have applied *MetaConfigurator* on several real-world schemas (such as EnzymeML [41] and the Strenda schema [42]) and configuration files from the real-world, to verify that it works and does bring benefits to the user.

We divide our interview tasks into four parts:

- Setup: We guide the participant through the setup of the tool, which involves accessing the tool via a web browser and loading the example files.
- Information Retrieval Questions: with increasing difficulty, the participant has to retrieve information from the configuration file.
- Configuration Modifications: The participant has to modify the given configuration file in various ways.
- Schema modifications: The participant has to modify the schema file.

The interview tasks and additional files can be found in out GitHub repository[1].

[1] `https://github.com/PaulBredl/meta-configurator/tree/main/paper/user_study`

Interview Process We performed the interview with one interviewee each. The interview lasts around one hour. At the beginning, we ask the interviewee about approval of recording, and they can stop participating at any time. We introduce *MetaConfigurator* to the interviewee. Then we send the participant the tasks, an example configuration file and let them work on the tasks while sharing their screen. During this task-solving session, we provide the interviewee with some basic help if they ask specific questions about the tool. The interview is recorded, and we make notes of the answers, feedback, and behavior of the interviewee. Finally, in an open dialog, we ask the interviewee for more feedback and improvement suggestions as well as their opinion on the tool. After the interview, we ask each interviewee to fill out a survey with some questions about their background and their opinion on the tool.

6.3 Results

This section presents the results of the user study for each research question.

RQ1: Which aspects of the tool can be improved? Tables 7-11 (in the appendix) show the feedback of the interviewees, as well as which measures we took based on it.

RQ2: Are users able to perform the following types of tasks using the tool? Table 5 shows the accuracy and difficulties that the participants had when solving the tasks. Accuracy is determined as the ratio of tasks that were solved correctly without the need for any hints to the total number of tasks. We note that all participants were able to solve all tasks after receiving hints from the interviewer.

- **RQ2.1**: Table 5 shows that all participants were able to retrieve information related to the schema and data. All participants were using mostly the GUI panel to retrieve information. Difficulties occurred when participants did not know that they could use the tooltips to get more information about the properties.
- **RQ2.2**: All participants were able to modify the given test file according to the given tasks. For some (User study 1 and 4) it was challenging to find a particular property, and they were not aware of the search functionality.
- **RQ2.3**: Most of the users were able to modify the schema file according to the tasks. The biggest difficulty was that participants did not know how to add a new property in the schema editor using the GUI panel.

Given that the average accuracy is approximately 0.9, we conclude that the participants were able to perform all three types of tasks using *MetaConfigurator*.

Table 5
User Study - Task solving accuracy and difficulties

	Accuracy & Notes	Difficulties
User Study 1	Accuracy: 100%(11/11) Effective use of tooltips Used both GUI and code editor	Task 3.2: Setting Vehicle Type to the highest level: Took some time to find the field
User Study 2	Accuracy: 91%(10/11) Used search functionality Used both GUI and code editor	Task 4.2: Adding new property: Could not find where to add a new property in Schema Editor Did not know how to edit the property name
User Study 3	Accuracy: 82%(9/11) Effective use of tooltips Not familiar with JSON schema Used only code editor in the beginning and then only GUI editor after task 2.3.	Task 2.3: Duration of Simulation: Did not think about using the GUI panel to retrieve information Task 2.4: Validity of humidity: Mistakenly thought the humidity value was valid. Did not consider red underline as an error Task 4.2: Add new property: Tried to set the property name in the wrong place
User Study 4	Accuracy: 82%(9/11) Solved tasks in a short time Used only the GUI editor	Task 2.4: Validity of humidity: Mistakenly thought the humidity value was valid Did not find out that he could use tooltips Task 3.2: Setting Vehicle Type to the highest level: Did not scroll down in Dropdown menu Task 3.3: Validation Errors: Thought the red underline was spell-checking
User Study 5	Accuracy: 100%(11/11) Solved tasks in a short time Used both GUI and code editor	Task 3.2: Setting Vehicle Type to the highest level: Took some time to find the vehicle type Task 4.2: Adding new property: Did not set the property type

RQ3: Would people use the tool in practice? Table 6 shows the positive feedback that the participants provided and where they can imagine the tool to be used in practice. Three of the participants highlighted the intuitiveness of *MetaConfigurator*. Four of the participants explicitly describe the tool with the words “useful” or “helpful”. Additionally, all five participants responded

that they would use the tool. Consequently, this result suggests that there is a demand for *MetaConfigurator* in practice.

Note that we only include positive feedback in this table since criticism is already addressed in section 6.3.

6.4 Threats to Validity

Our user study is qualitative and only has a small sample size of 5 participants. Hence, the results are not representative and cannot be generalized. In addition, the tasks of the user study only cover a small subset of operations that a user may want to perform with *MetaConfigurator* to edit configuration files and schemas. Due to the time constraints of the user study, we only cover rather simple tasks. Thus, for very complex tasks, the user study is not conclusive.

To get feedback on the tool and find out whether it is useful in practice, these limitations are acceptable. For RQ2, we cannot generalize our results but only provide a first impression of how users work with *MetaConfigurator*.

7 Discussion

This section discusses the implications of our work and future work.

7.1 Implications of our Work

MetaConfigurator provides a novel approach for editing configuration files by combining the advantages of a GUI and a code editor. Our user study suggests that *MetaConfigurator* allows users to successfully modify files within the constraints of a schema and to modify the schema itself. The participants rated our tool as intuitive to use and responded that they would use it themselves.

However, the user study also revealed some limitations of *MetaConfigurator*. Editing schemas with *MetaConfigurator* is not as intuitive as editing configuration files, especially for users who are not familiar with JSON schema. JSON schema may be feature-rich and expressive, but it is also complex and hard to understand for new users. The next section discusses how future work might address these limitations.

7.2 Future Work

To make *MetaConfigurator* more useful for users who are not familiar with JSON schema, there are several possible improvements. First, a visual schema editor could be added to *MetaConfigurator*, similar to schema editors discussed in section 2.2. These would provide a graph view of the schema, which is easier to understand than the JSON schema in text form or the tree view in *MetaConfigurator*. Second, *MetaConfigurator* could provide more guidance for users who are not familiar with JSON schema, e.g., by providing an interactive tutorial or supporting a less complex schema language.

There are many other possible improvements to *MetaConfigurator*. A desktop version of *MetaConfigurator* could be developed, which would allow users to edit files on their local machine, which is more convenient than loading them into the web application. Similarly, integration into other tools, such as IDEs, could be helpful for many users. *MetaConfigurator* currently only supports JSON schema draft 2020–12, but it could be extended to support other drafts by converting imported schemas to the latest draft. Furthermore, YAML is not fully supported yet, which could be added in the future. Another point that can be addressed is the loss of formatting and comments in YAML documents when they are updated with new data. This could be avoided by replacing only the section in the YAML document that corresponds to the change, instead of replacing the complete document. To allow for different styles of formatting, the user could be provided with global formatting style settings (such as level of indentation or whether in YAML strings should be in quotation marks or not). To deal with the loss of comments, a technique that keeps track of any comments in the text and then restores them after the text is replaced could be implemented. This has already been done in another tool of one of the authors [36].

Finally, *MetaConfigurator* could be extended to support code generation, e.g., for generating Java classes from a JSON schema, which is useful for developers.

To improve the user study, it could be repeated with more participants, so that the results are more representative. Instead of just having participants solve tasks, it could also be interesting to have one group of participants solve tasks with *MetaConfigurator* and another group solve the same tasks with only a text editor. This way, we could evaluate whether *MetaConfigurator* is more efficient than just using a text editor.

8 Conclusion

This paper addresses the development of *MetaConfigurator*, a tool that generates YAML/JSON file editor GUIs tailored to a given data schema. Our approach allows users to edit configuration files in a GUI, while still having the flexibility and speed of a code editor. Additionally, it removes the need for developing and maintaining a custom GUI for a particular schema. We use JSON schema as a schema language for the tool as it is an expressive and popular schema language. The tool is successfully implemented and our user study shows that it can be applied to solve practical tasks in 1) retrieving information from configuration files in the context of a schema, in 2) modifying configuration files, and in 3) editing schemas. The interest and positive feedback of the participants suggests that *MetaConfigurator* will be used in practice.

References

1. JSON Schema — json-schema.org. `https://json-schema.org`, [Accessed 01-05-2023]

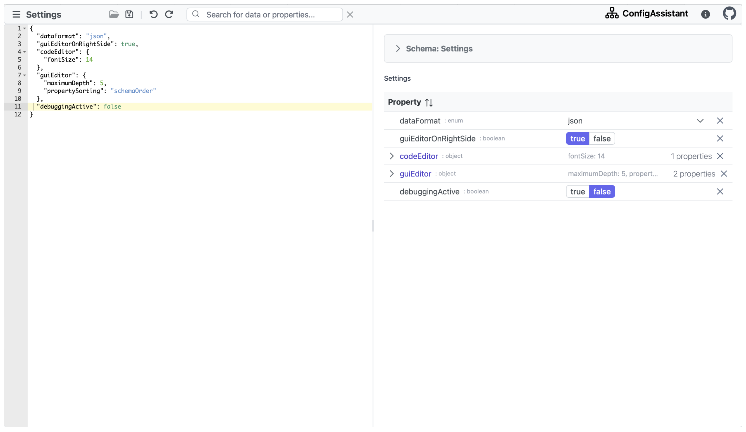
2. JSON Schema editor for Windows — json-buddy.com. https://www.json-buddy.com/json-schema-editor.htm, [Accessed 08-10-2023]
3. Leading Serialization Format for Record Data, https://www.ibm.com/topics/avro, [Accessed 14-06-2023]
4. Protocol Buffers — protobuf.dev. https://protobuf.dev, [Accessed 01-05-2023]
5. The complete solution for XML authoring, development and collaboration. — oxygenxml.com. https://www.oxygenxml.com/xml_developer.html, [Accessed 08-10-2023]
6. XML Editor - XMLBlueprint — xmlblueprint.com. https://www.xmlblueprint.com, [Accessed 08-10-2023]
7. XML editor and validator tool — xml-buddy.com. https://www.xml-buddy.com, [Accessed 08-10-2023]
8. XML Editor: XMLSpy — altova.com. https://www.altova.com/xmlspy-xml-editor, [Accessed 08-10-2023]
9. Configure Unify Execute (2019), https://cuelang.org/, [Accessed 18-05-2023]
10. A. Wright, H. Andrews, B.H.: JSON Schema Validation: A Vocabulary for Structural Validation of JSON (March 20, 2020), https://json-schema.org/draft/2019-09/json-schema-validation.html, [Accessed 06-05-2023]
11. A.Wright, H.Andrews Ed, B.H.: Validation (18 December 2022), https://json-schema.org/draft/2020-12/json-schema-validation.html#name-validation-keywords-for-any, [Accessed 18-05-2023]
12. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C., Scherzinger, S.: An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents (Long Version) (2021)
13. baeldung: Validate an XML file against an XSD file (Sep 2023), https://www.baeldung.com/java-validate-xml-xsd
14. Bex, G.J., Neven, F., Van den Bussche, J.: DTDs versus XML schema: A Practical Study. In: Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004. p. 79–84. WebDB '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1017074.1017095, https://doi.org/10.1145/1017074.1017095
15. Bosak, J., Bray, T., Connolly, D., Maler, E., Nicol, G., Sperberg-McQueen, M., Wood, L., Clark, J.: W3C XML Specification DTD ("XMLspec") (1998), https://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm
16. Carbonnelle, P.: TOP IDE Top Integrated Development Environment index — pypl.github.io. https://pypl.github.io/IDE.html, [Accessed 18-05-2023]
17. Carion, U.: JSON Type Definition. RFC 8927 (Nov 2020). https://doi.org/10.17487/RFC8927, https://www.rfc-editor.org/info/rfc8927
18. Deligiannidis, L., Kochut, K.J., Sheth, A.P.: RDF Data Exploration and Visualization. In: Proceedings of the ACM First Workshop on CyberInfrastructure: Information Management in EScience. p. 39–46. CIMS '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1317353.1317362, https://doi.org/10.1145/1317353.1317362
19. etc., E.P.: A useful library for validating a JSON file based on the schema (May 17, 2015), https://ajv.js.org/
20. Fallside, D., Walmsley, P.: XML Schema Part 0: Primer Second Edition - W3C Recommendation 28 October 2004 (2004), https://www.w3.org/TR/2004/REC-xmlschema-0-20041028/
21. Frasincar, F., Telea, A., Houben, G.J.: Adapting Graph Visualization Techniques for the Visualization of RDF Data, pp. 154–171. Springer London, Lon-

don (2006). https://doi.org/10.1007/1-84628-290-X_9, https://doi.org/10.1007/1-84628-290-X_9
22. Hansen, M.: GitHub - mokkabonna/json-schema-merge-allof: Simplify your schema by combining allOf — github.com. https://github.com/mokkabonna/json-schema-merge-allof (2023), [Accessed 18-10-2023]
23. Hartig, O., Hidders, J.: Defining Schemas for Property Graphs by Using the Graphql Schema Definition Language. In: Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). GRADES-NDA'19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3327964.3328495, https://doi.org/10.1145/3327964.3328495
24. Harutyun Amirjanyan, F.J.: Code Editor (Mar 28, 2010), https://ace.c9.io/, [Accessed 14-05-2023]
25. Higgins, D., Icebob: GitHub - vue-generators/vue-form-generator: :clipboard: A schema-based form generator component for Vue.js — github.com. https://github.com/vue-generators/vue-form-generator (2019), [Accessed 08-10-2023]
26. Jensen, D., Bennett, M.J., Dervisevic, D., Edwards, C., Marcacci, M.: GitHub - json-schema-form/angular-schema-form: Generate forms from a JSON schema, with AngularJS! — github.com. https://github.com/json-schema-form/angular-schema-form (2016), [Accessed 08-10-2023]
27. de Jong, J.: JSON Editor Online: JSON editor, JSON formatter, query JSON — jsoneditoronline.org. https://jsoneditoronline.org, [Accessed 08-10-2023]
28. Kappestein, C.: Typeschema (2023), https://typeschema.org/
29. Kristensen, M.: JSON Schema Store — schemastore.org. https://www.schemastore.org/json/, [Accessed 07-10-2023]
30. Küçükoğlu, T.: PrimeVue — Vue UI Component Library — primevue.org. https://primevue.org (2023), [Accessed 18-10-2023]
31. Lee, D., Chu, W.W.: Comparative Analysis of Six XML Schema Languages. SIGMOD Rec. **29**(3), 76–87 (sep 2000). https://doi.org/10.1145/362084.362140, https://doi.org/10.1145/362084.362140
32. Limited, L.T.: JSON Schema Editor — liquid-technologies.com. https://www.liquid-technologies.com/json-schema-editor, [Accessed 08-10-2023]
33. Marrs, T.: JSON at work: practical data integration for the web. O'Reilly Media, Inc. (2017)
34. Martens, W., Neven, F., Niewerth, M., Schwentick, T.: Bonxai: Combining the Simplicity of DTD with the Expressiveness of XML schema. ACM Trans. Database Syst. **42**(3) (aug 2017). https://doi.org/10.1145/3105960, https://doi.org/10.1145/3105960
35. Müller, E., Neufeld, E., Dirix, S., Koehler, L., Gareis, F.: More forms. Less code. - JSON Forms — jsonforms.io. https://jsonforms.io (2021), [Accessed 08-10-2023]
36. Neubauer, F.: GitHub - Logende/Bossshopproeditor: powerful and user-friendly web application, which makes setting up shops for the bukkit plugin BossShopPro way easier. — github.com. https://github.com/Logende/BossShopProEditor (2023), [Accessed 11-10-2023]
37. Neubauer, F.: Bossshoppro - the most powerful chest GUI shop/menu plugin., https://www.spigotmc.org/resources/bossshoppro-the-most-powerful-chest-gui-shop-menu-plugin.222/
38. North, C., Conklin, N., Saini, V.: Visualization schemas for flexible information visualization. In: IEEE Symposium on Information Visualization, 2002. INFOVIS 2002. pp. 15–22 (2002). https://doi.org/10.1109/INFVIS.2002.1173142

39. Perriault, N., Ramaswami, A., Grosenbacher, N.: GitHub - rjsf-team/react-jsonschema-form: A React component for building Web forms from JSON Schema. — github.com. `https://github.com/rjsf-team/react-jsonschema-form` (2023), [Accessed 08-10-2023]
40. Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D.: Foundations of JSON schema. In: Proceedings of the 25th International Conference on World Wide Web. p. 263–273. WWW '16, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE (2016). `https://doi.org/10.1145/2872427.2883029`, `https://doi.org/10.1145/2872427.2883029`
41. Range, J.: GitHub - Lightweight syntax for rapid development of data management solution in enzymology and biocatalysis. (2020), `https://github.com/EnzymeML/PyEnzyme`, [Accessed 19-10-2023]
42. Range, J.: GitHub - Standards for Reporting Enzymology Data. (2023), `https://github.com/EnzymeML/strenda-specifications/blob/master/specifications/strenda-specs.md`, [Accessed 19-10-2023]
43. Siffa, I.C., Schäfer, J., Becker, M.M.: Adamant: a JSON schema-based metadata editor for research data management workflows. F1000Research **11** (2022)
44. Silva, I.C.S., Santucci, G., Freitas, C.M.D.S.: Visualization and analysis of schema and instances of ontologies for improving user tasks and knowledge discovery. Journal of Computer Languages **51**, 28–47 (2019). `https://doi.org/https://doi.org/10.1016/j.cola.2019.01.004`, `https://www.sciencedirect.com/science/article/pii/S1045926X17302458`
45. You, E.: Vue.js - The Progressive JavaScript Framework — Vue.js — vuejs.org. `https://vuejs.org` (2023), [Accessed 18-10-2023]
46. Zimmer, S., Chapellier, C., Daoust, F.: GitHub - jsonform/jsonform: Build forms from JSON Schema. Easily template-able. Compatible with Bootstrap 3 out of the box. — github.com. `https://github.com/jsonform/jsonform` (2021), [Accessed 08-10-2023]

9 Design Appendix

10 User Study Appendix



```
{
  "dataFormat": "json",
  "guiEditorOnRightSide": true,
  "codeEditor": {
    "fontSize": 14
  },
  "guiEditor": {
    "maximumDepth": 5,
    "propertySorting": "schemaOrder"
  },
  "debuggingActive": false
}
```

Settings

Property		
dataFormat : enum	json	
guiEditorOnRightSide : boolean	[✓ true] false	
codeEditor : object	fontSize: 14	1 properties
guiEditor : object	maximumDepth: 5, propert...	2 properties
debuggingActive : boolean	true [✓ false]	

Fig. 13: UI of the Settings view

Table 6
User Study - Would people use the tool in practice?

User	Profession	Use Tool?	Potential Use Case	Positive Feedback
1	Professor	Yes	Configuring CI and simulations	“[I think always having those two views (code panel and GUI panel) is very helpful].” “[Modifying things using the GUI panel felt more comfortable than the code panel].” “Nice examples shown. [Without the GUI] you’d need to jump to the schema to find the max value.”
2	Professor	Yes	“I’d especially recommend [the tool] to some project partners.” “I’d certainly invite another partner to join in.”	“Very intuitive already.” “The search [functionality] is good.” “I like the tool; working with the GUI panel is intuitive and having an overview in the [code panel] helps. The combination of both is helpful and really good.”
3	Software Engineer	Yes	“Working with experimental partners, data often isn’t machine-readable. [. . .] A tool like this, where scientists define properties and schema, would be a huge advantage.”	“The tool is really, really nice and will help many people organize their data.” “Having a clickable, intuitive way to show schema structure is neat and looks pretty.” “I see potential impact in our research field; promoting this to partners could make a difference.”
4	Professor	Yes	Chemistry and biology field	We only discussed on a technical level; no general feedback collected.
5	Student	Yes	“An interface like this is helpful for JSON edits, especially with error indication.” “I’m a fan; even though I use Git on the command line, GUI tools are faster for many tasks, and your tool feels similar.”	“[I was] positively surprised; it was easy to navigate. Parsing JSON manually is a pain; this tool felt like Word compared to LaTeX, allowing intuitive changes directly in the interface.” “Are you going to open source this? It could be useful.”

Table 7
User Study 1 - Feedback and Solution

Feedback	Solution
The property value should not be autocorrected if the user enters an incorrect value. Instead, an error message or another way should be used to inform the user that their input is incorrect.	Instead of autocorrecting values, we now provide more clear user feedback on incorrect values (red underline, error symbol).
It would be good to have the ability to remove data entries with the GUI panel.	Implemented by adding a *remove* button next to properties that have data and are not required.
A search functionality to locate properties would be helpful, especially within nested levels.	Implemented in the toolbar. All findings are highlighted in the GUI panel.
The GUI panel feels overwhelming to the user due to many variations in the styling and color of the GUI elements.	We slightly reduced the number of different styles by no longer showing required properties in boldface and instead just showing an asterisk next to it.
The cursor should not have the clickable animation when hovering over non-clickable fields in the GUI editor.	Now we only show the clickable animation when hovering over clickable GUI components.
In drop-down menus, we do not need a button to clear the selection.	We disabled the option of clearing the selection.
If the type of a property is “any”, it should not be interpreted as the “string” type in the GUI panel.	We improved the representation of the “any” type, but further improvements are needed, which will be considered in future work.
Validation errors should not be highlighted via a warning symbol, but instead an error symbol.	We changed the warning symbol into an error symbol.
After performing an undo or redo action, the cursor should jump to the corresponding location to reflect the changes made by the user.	This will be considered in future work.

Table 8
User Study 2 - Feedback and Solution

Feedback	Solution
A graph-based view would be more intuitive for handling complex data structures.	This will be considered in future work.
Providing immediate feedback to users when they enter incorrect ranges is essential to prevent them from inputting invalid values into the property.	We now highlight schema violations by a red error symbol in the GUI panel and underlining the property name in red. Additionally, the tooltip lists all schema violations of a property.
Validation errors should also be reflected in the GUI panel, including for child properties.	See the point above. Also, now the tooltip lists schema violations of child properties.
When dealing with an array, the display name of array elements (index) should be improved. Currently, the tool only shows the element index.	We replaced the numerical labels with a standard programming notation, which is `propertyName[0]`, `propertyName[1]`, ...
The input field next to the *Add Item* button is confusing. Both the input field and the button can be used to create a new item, which is redundant.	We removed the input field next to the button.
It would be more consistent if all user input in the GUI panel was within the right column of the table. In some scenarios user input is needed within the left column (for names of new properties), which feels inconsistent.	Because of the nature of JSON schema, we retained the property name within the left column. To make it clear to the user that the property name can be edited, we added an *edit* icon next to it.
The search function for locating specific properties lacks clarity at first glance. It should provide an immediate response and extend to nested levels, rather than merely highlighting the higher-level findings.	The search now provides a list of results, and upon clicking on a particular result, it jumps to that result in the code panel and GUI panel. In the GUI panel, if the element is nested, its parents will be automatically expanded.

Table 9
User Study 3 - Feedback and Solution

Feedback	Solution
Working with the schema editor is difficult for me. It does not feel intuitive.	We made the schema editor more intuitive by creating our own simplified JSON schema meta schema. For example, advanced JSON schema features are separated from the simple ones. See section 5.5

Table 10
User Study 4 - Feedback and Solution

Feedback	Solution
Modifying or renaming a new property in the GUI panel does not appear to take effect when double-clicking on it.	Renaming properties in the GUI panel can now be done using the *edit* button next to the property name.
When creating a new property in the schema editor, its sub-schema has to be selected, such as *string property* or *boolean property*. Additionally, the type of the property has to be selected by the user too. Therefore, for example, when creating a new *string property*, the user has to select that it is a string two times. It would be much more intuitive if the selection needs to be done only one time.	We completely overhauled our JSON schema meta schema. Now, when creating a new property, the user will have to select the type only once.
A toggle button should be implemented to enable and disable the code panel and GUI panel.	Only having a GUI panel or only having a code panel restricts the user unnecessarily. The interplay of both panels is what makes this tool most effective. If the user does not want to use one of the panels, they can resize that panel to a very small size.
When working with a particular property in the GUI panel, the opacity of the other properties should be decreased, visually highlighting the currently focused property.	Will be considered in future work.
Simplify the schema editor to make it easier to work with, for those who are not very familiar with JSON schema.	Has been done, see section 5.5.

Table 11
User Study 5 - Feedback and Solution

Feedback	Solution
The search button is not immediately evident, making it challenging for users to locate the search function.	Instead of showing the search bar only when clicking the search button, we now always show it.