

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

Santosh Kumar
santosh.research888@gmail.com
Independent Researcher

Abstract

This paper presents MetaConfigurator, a tool that generates customizable graphical user interfaces (GUIs) for editing YAML and JSON configuration files based on a provided data schema. By leveraging the expressiveness of JSON Schema (draft 2020-12), MetaConfigurator offers a flexible and efficient way to edit configuration files while maintaining the speed and control of traditional code editors. The tool reduces 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.

Keywords

• MetaConfigurator • JSON Schema • YAML/JSON Configuration Files • Graphical User Interface (GUI) Generation • Schema-Aware Editing • Configuration Management

1. Introduction

Structured data formats such as JSON, XML, and YAML are widely used in software development and scientific research for configuration files, data exchange, and structured data storage, owing to their dual human-readable and machine-parsable nature. The structure of such data can be formally specified by schemas, which define the rules that valid data must satisfy. Given a schema, one can validate whether a particular document conforms to it. Throughout this paper, we refer to any concrete YAML or JSON document that conforms to a given schema as a *configuration file* ().

Depending on the application domain, can be complex and time-consuming to edit and maintain. For instance, a configuration file for a scientific simulation or cloud infrastructure may contain hundreds of parameters with intricate dependencies and constraints. Manual editing of such files is error-prone and demands detailed knowledge of the underlying schema. The breadth of applications relying on such configuration files is immense. Domains as diverse as protein dynamics analysis [1], statistical data visualization [2], and SQL-based system optimization [3] all depend on structured configuration files. Other examples include association rule mining [4], energy consumption analytics [5], population genomics [6], cloud SQL migration [7], password-less authentication [8], project management [9], wildfire prediction, distributed data analytics [10], networked filesystem benchmarking [11], kernel-level

data probes [12], linear algebra kernel synthesis [13], and engineer-task matching [14]. Graphical user interfaces (GUIs) can mitigate this burden by offering intuitive form fields, dropdown menus, and immediate validation feedback. However, building and maintaining bespoke GUIs requires substantial initial development effort and continuous updates whenever the schema evolves, which is a barrier for projects with limited resources or rapidly changing schemas.

We address this problem by developing MetaConfigurator, a web application that automatically generates such GUIs based on a provided data schema. The tool eliminates the need for hand-crafted interfaces by interpreting the schema and dynamically constructing an appropriate GUI. It combines the efficiency of a text editor with the assistance of a GUI, allowing users to choose the mode that best fits their task and expertise. Additionally, the schema itself can be edited using the same tool, thanks to a self-referential meta-schema approach.

Our approach differs from other schema-to-UI approaches in the following ways:

1. The tool combines the support of a GUI with the flexibility and speed of a code editor by providing both in one view. The code editor offers syntax highlighting, auto-completion, and error highlighting, making it suitable for experienced users who prefer direct text manipulation.
2. The schema can be edited using the same tool and type of view. We use the meta-schema of the schema language to generate a GUI for editing the schema. This means the user experiences a consistent interface whether editing data or schema, lowering the learning curve.
3. We support advanced mapping features such as conditional logic (if-then-else), dependent schemas, and constraints, which are often crucial for real-world configuration files.

The remainder of this paper is organized as follows. Section 2 discusses related work on schema languages and schema-to-UI generation. Section 3 evaluates existing schema languages and justifies our choice of JSON Schema. Section 4 presents the design and architecture of MetaConfigurator, including the user interface and data flow. Section 5 covers implementation details, focusing on schema preprocessing and the custom meta-schema. Section 6 describes a qualitative user study conducted to evaluate the tool's usability and effectiveness. Section 7 discusses implications, limitations, and future work. Finally, Section 8 concludes the paper.

2. Related Work

This section reviews existing schema languages and approaches to generate UIs from schemas. Because our research is practical in nature, we also consider non-academic sources such as specifications and tool documentation.

2.1 Schema Languages

Schema languages are formal languages that define the structure, constraints, and relationships of data. Since our tool generates a GUI based on a schema, choosing an appropriate schema language is crucial. We now describe several existing schema languages and later evaluate them to determine the most suitable one.

JSON Schema is a vocabulary that allows annotation and validation of JSON documents. It has become the de facto standard for describing JSON data, with wide adoption in web APIs, configuration files, and NoSQL databases. JSON Schema supports simple types (string, number, integer, boolean),

complex types (object, array), and a rich set of validation keywords such as `enum`, `const`, `minimum`, `pattern`, and conditional constructs like `if/then/else`. JSON Schema can also be applied to YAML because YAML is a superset of JSON, although some YAML-specific features (e.g., anchors, custom tags) cannot be expressed. The JSON Schema Store provides hundreds of ready-made schemas for popular configuration file types, including Docker Compose, GitHub Actions, and OpenAPI specifications. Recent research has advanced JSON Schema handling, including algorithms for schema inference [15], witness generation [16, 17], and extraction of tagged unions [18].

XSD (XML Schema Definition) and **DTD** (Document Type Definition) are schema languages for XML. XSD is more expressive and modern, supporting namespaces, data types, and complex content models. DTD is older and simpler, offering basic element and attribute declarations. W3C recommends XSD as the schema language for XML documents.

Other schema languages we considered include:

- **CUE** (Configure, Unify, Execute): an open-source data validation language that works with JSON and YAML, and is a superset of both. It emphasizes unification and constraints.
- **Apache Avro**: a data serialization framework that uses a JSON-based schema language, often used in big data pipelines.
- **JSON Type Definition (JTD)**: a simpler schema language for JSON, designed to be easier to understand and implement than JSON Schema, but less expressive.
- **TypeSchema**: a schema language for JSON documents, similar to JTD but with different syntax and some additional features.
- **Protocol Buffers**: a language-neutral, platform-neutral extensible mechanism for serializing structured data, developed by Google.
- **GraphQL schema language**: used to define the structure of GraphQL APIs, focusing on types and fields.

We do not consider graphical modeling languages like UML or ER diagrams because they are not text-based and not directly suitable for validation. Likewise, ontology languages like OWL or RDF Schema are not intended for data validation but for knowledge representation.

2.2 Existing Approaches

Many tools validate against a schema, typically within IDEs or libraries. For example, Visual Studio Code and IntelliJ IDEA can validate JSON and YAML files against a provided schema, offering error messages, auto-completion, and syntax highlighting. However, they do not provide a GUI for editing the data based on the schema; the user still works primarily with raw text.

Form generators produce web forms from a schema. Examples include React JSON Schema Form, Angular Schema Form, Vue Form Generator, JSON Forms, JSON Editor, and JSON Form. These tools are based on JSON Schema and create a form that the user fills out; the resulting JSON document is then validated against the schema. They provide specific components for each data type (e.g., text field for strings, number field for numbers, dropdown for enums). However, most require an additional "UI schema" to customize the generated form, which itself must be created and maintained. Moreover, they do not offer a text-based editor, limiting their appeal for experienced users who prefer to edit data directly.

Adamant is a JSON-schema-based form generator designed specifically for scientific data. It supports a subset of JSON Schema and allows editing and creating JSON schema files, with separate modes for schema editing and data editing. It extracts units from field descriptions, which is helpful for scientific applications. Limitations include the lack of a text editor and limited support for the full JSON Schema specification. The design of generated GUIs is an active research area, with studies on combinatorial optimization of layouts [19], user experience in web map interfaces [20], and domain-specific GUI design for energy management [21]. Our tool differs from these form generators by combining a GUI with a code editor and supporting schema editing in the same interface. A preliminary version of MetaConfigurator was described in [22], and subsequent extensions have connected it to the Semantic Web [23] and explored conversions between SHACL and JSON Schema [24].

Schema editors, such as JSON Editor Online and commercial tools like Altova XMLSpy and Liquid Studio, focus on editing schemas themselves. They may provide text-based or graphical views but do not emphasize assisting with data editing based on the schema.

Our tool, MetaConfigurator, differs by combining a GUI and a code editor in a single view, supporting both data and schema editing with the same interface, and being generic enough to handle any JSON Schema without requiring additional UI schemas. This makes it a more versatile and user-friendly solution for a broad range of users.

3. Evaluation of Schema Languages

To select the most suitable schema language for MetaConfigurator, we evaluated the languages based on practical usage and expressiveness.

3.1 Evaluation Criteria

Practical usage was measured by:

- The number of Stack Overflow results (via Google search with “site:stackoverflow.com”), retrieved in January 2026, which indicates community adoption and the likelihood of finding help.
- IDE support: checking the 10 most popular IDEs for native or plugin support for the language.
- Library support: number of Node.js packages on npmjs.com related to the language, which is important for our web-based tool.

Expressiveness was assessed against nine features that we consider important for our approach:

1. Simple types (string, number, boolean, integer)
2. Complex types (object, array)
3. Descriptions (for tooltips and help)
4. Example values (placeholders)
5. Default values
6. Optional/required distinction
7. Constraints (length, range, enumeration, pattern)

8. Conditional dependencies (if-then-else)
9. References (reusable sub-schemas)

Each feature was marked as supported or not.

3.2 Evaluation Results

Table 1 summarizes the results. JSON Schema stands out as both highly popular (245,000 search results, 8/10 IDE support, 4,536 Node packages) and fully expressive (9/9 features). XSD is also popular and expressive (8/9) but is tied to XML, which is less common for modern configuration files. DTD, CUE, and others are either less popular or less expressive. Therefore, we selected JSON Schema (draft 2020-12) as the schema language for MetaConfigurator.

The evaluation results directly inform our design decisions. The high popularity and broad IDE/library support of JSON Schema ensure that users can readily find help and integrate MetaConfigurator with existing workflows. Its full expressiveness, including conditional logic and references, enables the tool to handle real-world configuration files without requiring a simplified subset. In contrast, XSD's tie to XML would restrict the tool's applicability to modern YAML/JSON ecosystems, while languages like CUE, despite good expressiveness, lack the same level of community adoption and tooling, which would increase the learning curve for new users. The choice of JSON Schema thus aligns with our goal of creating a practical, schema-aware editor that works out of the box for a wide range of configuration formats.

Table 1: Evaluation of schema languages. Expressiveness is the number of features supported out of 9.

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

4. Design

4.1 User Interface Overview

The UI of MetaConfigurator has three views: *File Editor*, *Schema Editor*, and *Settings*. In the File Editor view (Figure 1), the user can edit their configuration file using both a code panel (left) and a GUI panel (right). As shown in Figure 1, the File Editor view consists of four main components: (1) the view switcher, (2) the toolbar, (3) the code panel, and (4) the GUI panel. The code panel provides syntax highlighting and schema validation, while the GUI panel offers form-like input elements based on the schema. The Schema Editor view (Figure 2) allows editing the schema itself using the same dual-panel approach, with the JSON Schema meta-schema as the underlying schema. The Settings view lets users adjust tool parameters such as indentation, font size, and default expansion depth.

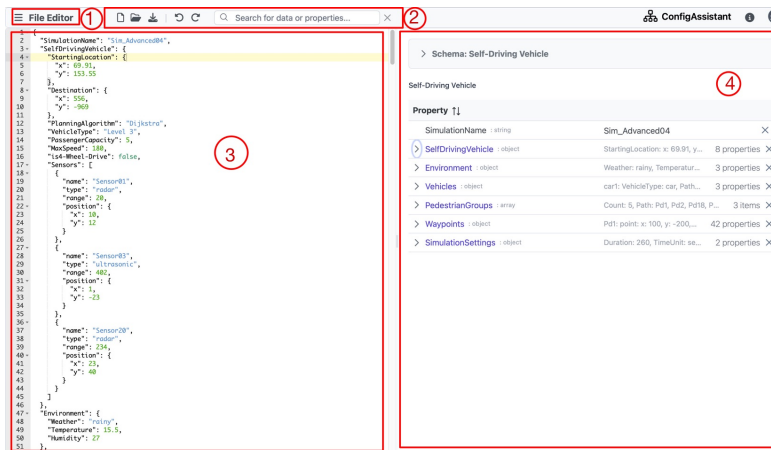


Figure 1: UI of file editor view. Different components highlighted: 1) view switcher, 2) toolbar, 3) code panel, 4) GUI panel.

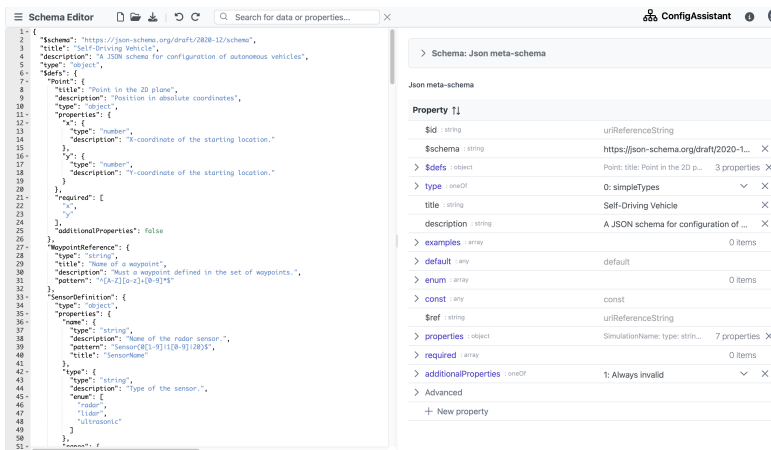


Figure 2: UI of schema editor view.

The GUI panel is generated by traversing the schema tree, where each schema node maps to a corresponding GUI element (e.g., a text field for strings, a checkbox for booleans, or a dropdown for enums). Complex nodes such as objects and arrays are expandable, allowing users to navigate nested structures. The tool shows only the top level by default, and the user can expand or zoom into nested properties. A breadcrumb shows the current path, allowing easy navigation back to parent levels.

4.2 Architecture

The core of MetaConfigurator is a single source of truth data store that holds the current configuration data as a JavaScript object. Editor panels (code editor, GUI panel) subscribe to this store and can update it. This decoupled architecture allows adding new panels (e.g., for other data formats) without affecting existing ones. The data store also stores the currently selected schema and the path of the currently focused element.

The code panel uses the Ace Editor library and provides syntax highlighting, code folding, and validation. It subscribes to the data store and synchronizes with it as follows:

- When the data store changes, the code panel serializes the JavaScript object into the chosen format (JSON or YAML) and replaces the text.

- When the user edits the text, it is parsed and sent to the data store, which then notifies all subscribed panels.

The GUI panel is built by mapping each schema node to a corresponding GUI component. For example, a string property becomes a text field, a boolean becomes a checkbox, and an enum becomes a dropdown. Tooltips provide additional schema information. Validation errors are highlighted with red underlines and icons.

4.3 Intended Workflow

The typical workflow is:

1. Upon first access, a dialog asks the user to select a schema (or load one from a file).
2. The GUI panel is automatically generated on the right side of the file editor.
3. The user can then create or modify with the help of the GUI, or edit the raw text in the code panel.
4. If the schema needs to be modified, the user switches to the schema editor, which provides the same dual-panel interface for editing the schema.
5. Changes to the schema automatically reflect in the file editor (e.g., new properties appear, old ones disappear).

In summary, the design of MetaConfigurator is guided by three principles: (i) the combination of a code editor and a generated GUI to cater to users with different expertise and preferences; (ii) the use of a single, decoupled data store to synchronize all views and allow future extension to other formats or panel types; and (iii) the automatic generation of the GUI entirely from the schema, eliminating the need for hand-crafted interface definitions. These principles keep the tool flexible, maintainable, and user-friendly.

5. Implementation

This section describes implementation details of MetaConfigurator.

5.1 Technologies

We use Vue.js as the UI framework and PrimeVue as the component library. We chose Vue.js for its reactive data binding, which simplifies synchronization between the data store and multiple editor panels, and PrimeVue for its comprehensive set of form components that map well to JSON Schema types. The code editor is Ace Editor, and JSON Schema validation is performed with Ajv, which supports draft 2020-12. The application is entirely client-side, running in the browser.

5.2 Code Panel

The code panel embeds Ace Editor with syntax highlighting. Schema validation errors are marked with red underlines and descriptions. The code panel supports bidirectional navigation: selecting a property in the GUI panel scrolls the code panel to the corresponding line, and vice versa. This linkage is currently implemented for JSON but not yet for YAML.

The code editor also supports drag-and-drop file opening, undo/redo, and font size adjustment. The editor can be configured to format the code automatically (pretty-print) on load or save.

In brief, the code panel provides a full-featured text editor with schema validation and synchronization with the data store, enabling direct manipulation of configuration files while receiving immediate schema feedback.

5.3 GUI Panel

The GUI panel presents the configuration data in a table-like structure where each row represents a key-value pair. Arrays are represented similarly, with indices as keys. The panel supports:

Overall, the GUI panel translates schema constraints into interactive form elements, guiding users toward valid data entry while still allowing the full expressive power of the underlying JSON Schema.

To illustrate how the components work together, consider a user editing a YAML configuration for a simulation. Upon loading the file and its JSON Schema, the GUI panel displays the top-level properties as rows. Suppose the property `simulationType` is an enum with values `deterministic` and `stochastic`; the user selects one from a dropdown. The schema may conditionally require a `seed` property only when `simulationType` is `stochastic`. In this case, the GUI panel automatically shows or hides the `seed` field based on the current selection. When the user changes the value of `seed` to a non-integer, the field is highlighted red and a tooltip explains the expected type. Simultaneously, the code panel updates to reflect the new YAML content, and if the user prefers, they can directly edit the text, with validation errors appearing as underlines. This unified workflow ensures that users can switch between GUI and code editing without losing context or consistency.

The correspondence between JSON Schema constructs and GUI components is summarized in Table 2.

Table 2: Corresponding GUI implementations for JSON Schema features.

JSON type/keyword	Schema	GUI implementation
string, number, integer		Text field (number/integer with increment/decrement buttons)
boolean		Checkbox
object, array		Expandable list of child elements (array has add button)
enum, const		Dropdown menu
required		Red asterisk
deprecated		Strikethrough property name
anyOf		Multiselect menu
oneOf		Dropdown menu

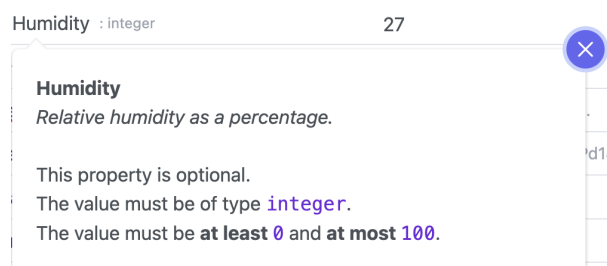


Figure 3: Tooltip showing schema information and validation errors.

As shown in Figure 3, tooltips display schema information and validation errors, helping users understand constraints and correct invalid entries.

5.4 Schema Preprocessing

To generate the GUI, the schema is preprocessed in three stages:

1. **One-time preprocessing** (when loading):
 - Inject property names as titles if missing.
 - Convert `const` to `enum` with a single value.
 - Deduce types from `enum` values if type is not specified.
2. **Lazy preprocessing** (when a node is expanded):
 - Resolve `$ref` references within the same file.
 - Merge `allOf` sub-schemas into one.
 - Convert type unions (e.g., `"type": ["object", "string"]`) to `oneOf`.
 - Remove incompatible `oneOf/anyOf` options.
 - Merge redundant `oneOf/anyOf` that have only one compatible option.
 - Attempt to merge `oneOf` into `anyOf` when possible.
3. **Effective schema calculation** (on each data change):
 - Evaluate `if/then/else` conditions.
 - Handle `dependentRequired` and `dependentSchemas`.

These preprocessing steps are only used for generating the GUI; the original schema file is not modified. As a result, the schema editor always displays the raw schema as provided by the user.

To summarize, the staged preprocessing pipeline transforms complex JSON Schema constructs into a simplified internal representation that can be directly mapped to GUI components, without altering the user's original schema.

5.5 Custom Meta Schema for Schema Editor

The official JSON Schema meta-schema is not user-friendly for GUI generation. We created a modified meta-schema that:

- Adds descriptions for all fields, making the purpose of each keyword clear.
- Removes external references and dynamic anchors (replaced with local `$ref`).
- Uses `if` conditions to show only context-relevant fields. For example, properties related to arrays are only shown when the type is `array`.
- Introduces an `advanced` section to hide rarely used keywords, based on an analysis of common schema usage. This reduces clutter and makes the editor more approachable for beginners.

Taken together, the custom meta-schema tailors the schema editing experience to the needs of both novice and advanced users by exposing only context-relevant keywords and providing descriptive help text.

6. User Study

We conducted a qualitative user study with five participants to evaluate the tool's usability and effectiveness.

6.1 Research Questions

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 in the context of a given schema.
 - **RQ2.2** Modify 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

Participants were given a made-up self-driving car simulation schema and data file. The tasks were designed to cover all three research question areas. For information retrieval, participants had to find specific values or determine whether certain data was valid according to the schema. For modification, they had to change existing values, add new array elements, and set properties to valid values. For schema modification, they had to add a new property, set its type, and adjust constraints. We observed their interaction and collected feedback via surveys and open discussion.

6.3 Results

Table 3 summarizes task accuracy and observed difficulties. The average accuracy was approximately 90%. All participants were able to complete tasks after minor hints. Key difficulties included:

- Finding how to add new properties in the schema editor.
- Not noticing validation error indications (red underlines).
- Being unfamiliar with JSON Schema keywords.
- Difficulty locating specific properties in the GUI (some participants did not use the search functionality).

Two participants (Users 3 and 4) struggled with determining the validity of the humidity property. This suggests that the current validation feedback red underlines and tooltips may not be sufficiently prominent or may not explain why a value is invalid. This finding directly motivated our proposal to enhance error visibility (Section 7).

All participants responded positively and stated they would use the tool. Feedback highlighted the intuitiveness of the combined GUI and code editor, usefulness for configuration management, and potential for scientific data organization. Quotes from participants include: "The tool is really, really nice and will help many people organize their data." and "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."

Table 3: User Study - Task solving accuracy and difficulties.

User	Accuracy	Difficulties
1	100%	Finding a specific property took time
2	91%	Adding new property; editing property name
3	82%	Validity of humidity; adding new property
4	82%	Validity of humidity; scrolling dropdown
5	100%	Adding new property; not setting type

6.4 Threats to Validity

The study is qualitative with a small sample size (5 participants), so results are not statistically generalizable. Tasks covered only a subset of possible operations, and complex scenarios were not tested. Therefore, the findings should be considered as initial insights rather than definitive conclusions.

7. Discussion and Future Work

The user study indicates that MetaConfigurator effectively supports editing and schemas. However, schema editing was less intuitive for users unfamiliar with JSON Schema. To address the difficulties observed in the user study particularly the challenges in adding new properties in the schema editor, noticing validation errors, and understanding JSON Schema keywords, future work could include:

- Adding a visual schema graph editor to make schema creation and modification more intuitive for non-experts.
- Providing an interactive tutorial that covers common schema keywords and GUI interactions, directly addressing the learning curve observed.
- Enhancing error visibility, for instance by adding prominent warning banners or animations, since participants sometimes missed red underlines.
- Developing a desktop version and IDE integration to lower the barrier to adoption.
- Extending support to other JSON Schema drafts and full YAML features, including preservation of formatting and comments.
- Code generation from schemas to further assist users who prefer to start from a template.
- Conducting a larger user study with a control group using only a text editor to quantify efficiency gains.

8. Conclusion

MetaConfigurator is a tool that generates a schema-aware GUI for editing YAML/JSON configuration files. It combines a code editor and a GUI panel, supports schema editing, and uses JSON Schema as its foundation. A user study demonstrated that participants could successfully perform data retrieval, data modification, and schema modification tasks. Feedback suggests the tool is intuitive and would be used in practice. Future enhancements will further improve usability and functionality.

References

- [1] V. S. S. R. Kuppili. PyEDA: A python-based tool for protein dynamics analysis using essential dynamics approach. In D. Sinwar, V. S. Dhaka, T. Perumal, and E. Vocaturo, editors, *Innovations in Computational Intelligence and Computer Vision, ICICV 2025*, volume 1786 of *Lecture Notes in Networks and Systems*, Cham, Switzerland, 2026. Springer. doi: 10.1007/978-3-032-14757-8_10.
- [2] B. Nare. Flexplot: Enhancing data visualization in statistical modeling with simplified graphical tools. In *Proc. 2025 Int. Conf. Computational Innovations and Sustainable Technologies (ICCIST)*, pages 1–12, Bangalore, India, 2025. doi: 10.1109/ICCIST67338.2025.11438734.
- [3] S. Vangala. An adaptive physical design optimization framework for SQL-based systems using incremental workload analysis and cost-driven configuration search. *International Journal of Intelligent Systems and Data Science*, 1(4), 2026.
- [4] P. Parashar. Enhancing association rule mining with the CGRG algorithm: A cluster-based approach. In *Proc. 2025 IEEE/ACIS 23rd Int. Conf. Software Engineering Research, Management and Applications (SERA)*, pages 379–392, Las Vegas, NV, USA, 2025. doi: 10.1109/SERA65747.2025.11154642.
- [5] O. Ladapo. Empowering energy efficiency: A real-time mobile analytics platform for intelligent consumption monitoring. *International Journal of Science and Applied Technology (IJSAT)*, 16(2), 2025.
- [6] D. G. Battina. Next-gen variant analysis: A scalable toolkit for population-level genomics. In T. Senjyu, M. Mahmud, and A. Joshi, editors, *Smart Trends in Computing and Communications, SmartCom 2026*, volume 2006 of *Lecture Notes in Networks and Systems*, Cham, Switzerland, 2026. Springer. doi: 10.1007/978-3-032-27702-2_38.
- [7] H. N. Beesabathuni. Azure-SQL AutoSizer: Privacy-aware performance-cost SKU mapping for SQL migrations. In *Proc. Int. Conf. Intelligent Systems for a Sustainable Future (ISSF 2026)*, pages 1043–1055, 2026. doi: 10.2991/978-94-6239-693-7_101.
- [8] S. S. Gujar. Exploring device fingerprinting for password-less authentication systems. In *Proc. 2024 Global Conf. Communications and Information Technologies (GCCIT)*, pages 1–7, Bangalore, India, 2024. doi: 10.1109/GCCIT63234.2024.10862062.
- [9] J. Joseph. Developing a conversation-based project management tool. *International Journal of Engineering Sciences & Research Technology*, 14(4):1–20, 2025. doi: 10.29121/ijesrtp.v14.i4.2025.1.
- [10] V. Kaliappan. Data system languages: An expressive framework for distributed heterogeneous data analytics. *ESP International Journal of Advancements in Science & Technology (ESP-IJAST)*, 4(1): 81–90, 2026.
- [11] P. Singh. Workload-driven perspectives on networked filesystems: Benchmarking NFS/SMB and version-level trade-offs. *International Journal of Advanced Research and Interdisciplinary Scientific Endeavours*, 3(4):975–981, 2025. doi: 10.61359/11.2206-2553.

- [12] M. Rengaraj. Kernel-safe data probes: Context-aware address tagging with on-demand translation for OS modules. *International Journal of Intelligent Systems and Data Science*, 1(3), 2026. doi: 10.67231/cqx40j38.
- [13] S. Shah. Architecture-aware synthesis of fused linear algebra kernels. *International Journal of Intelligent Systems and Data Science*, 1(2), 2026. doi: 10.67231/ced8hs42.
- [14] N. V. Konduri. Variance-aware sprint assignment: An interpretable projection pipeline for engineer-task matching. In *Proc. 2026 Int. Conf. Computing, Electronics & Communications Engineering (ICCECE)*, pages 1–5, Kolkata, India, 2026. doi: 10.1109/ICCECE69715.2026.11492942.
- [15] I. V. Latták and P. Koupil. A comparative analysis of JSON schema inference algorithms. In *Proc. ENASE*, pages 379–386, 2022.
- [16] M. A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, and S. Scherzinger. Not elimination and witness generation for JSON schema. arXiv preprint arXiv:2104.14828, 2021.
- [17] L. Attouche, M. A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, and S. Scherzinger. Witness generation for JSON schema. arXiv preprint arXiv:2202.12849, 2022.
- [18] S. Klessinger, M. Klettke, U. Störl, and S. Scherzinger. Extracting JSON schemas with tagged unions. arXiv preprint arXiv:2306.07085, 2023.
- [19] A. Oulasvirta, N. R. Dayama, M. Shiripour, M. John, and A. Karrenbauer. Combinatorial optimization of graphical user interface designs. *Proceedings of the IEEE*, 108(3):434–464, 2020.
- [20] P. Cybulski and T. Horbiński. User experience in using graphical user interfaces of web maps. *ISPRS International Journal of Geo-Information*, 9(7):412, 2020.
- [21] N. Aksoy and İ. Genç. Design of graphical user interface for artificial intelligence-based energy management system for microgrids. *Electrica*, 23(2):202–211, 2023.
- [22] F. Neubauer, P. Bredl, M. Xu, K. Patel, J. Pleiss, and B. Uekermann. MetaConfigurator: A user-friendly tool for editing structured data files. *Datenbank-Spektrum*, 24(2):161–169, 2024.
- [23] M. Jafarkhani. Connecting MetaConfigurator to the semantic web: RDF authoring, ontology integration, and knowledge graph support, 2026.
- [24] G. Chinnikkaramadom Govindan. Bridging SHACL and JSON schema: Design, implementation, and evaluation of bidirectional conversions for JSON and JSON-LD documents, 2026.