future internet

MDPI

Article

RestRho: A JSON-Based Domain-Specific Language for Designing and Developing RESTful APIs to Validate RhoArchitecture

Enrique Chavarriaga [1], Luis Rojas [2,*], Francy D. Rodríguez [3], Kat Sorbello [4] and Francisco Jurado [5,*]

[1] Departament I+D+i, UGround Global S.L., 28003 Madrid, Spain; echavarriaga@uground.com
[2] Facultad de Ingeniería, Universidad San Sebastián, Santiago 8420524, Chile
[3] Computer Engineering Department, Universidad Politécnica de Madrid, 28223 Madrid, Spain; francydiomar.rodriguez@upm.es
[4] The Stella Way, Brisbane 4118, Australia; kat@thestellaway.com
[5] Department of Computer Engineering, Universidad Autónoma de Madrid, 28049 Madrid, Spain
* Correspondence: luis.rojasp@uss.cl (L.R.); francisco.jurado@uam.es (F.J.)

Abstract

Domain-Specific Languages with JSON grammar (JSON-DSLs) are specialized programming languages tailored to specific problem domains, offering higher abstraction levels and simplifying software implementation through the JSON standard. RhoArchitecture is an approach for designing and executing JSON-DSLs, incorporating a modular programming model, a JSON-based evaluation engine, and an integrated web development environment. This paper presents RestRho, a RESTful NodeJS server developed using two JSON-DSLs designed with RhoArchitecture: SQLRho and DBRestRho. These languages enable declarative specification of database operations and HTTP requests, respectively, supporting modularity, reuse, and template-based transformations. We validate the RestRho implementation through a dual approach. First, we apply software metrics to assess code quality, maintainability, and complexity. Second, we conduct an empirical study involving 39 final-year computer engineering students, who completed 18 structured tasks and provided feedback via questionnaires. The results demonstrate the tool's usability, development efficiency, and potential for adoption in web application development.

Keywords: domain-specific languages; JSON; JSON-DSL; API RESTful; Rho architecture; templates engine

check for updates

Academic Editor: Paolo Bellavista

Received: 13 June 2025
Revised: 15 July 2025
Accepted: 21 July 2025
Published: 31 July 2025

Citation: Chavarriaga, E.; Rojas, L.; Rodríguez, F.D.; Sorbello, K.; Jurado, F. RestRho: A JSON-Based Domain-Specific Language for Designing and Developing RESTful APIs to Validate RhoArchitecture. *Future Internet* **2025**, *17*, 346. https://doi.org/10.3390/fi17080346

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Nowadays, the efficient creation of web applications has become an increasingly crucial challenge. The demand for tools that streamline this process, enabling software engineers and domain experts to build web applications quickly and effectively, has led to the emergence of innovative approaches. In this context, Domain-Specific Languages (DSLs) have arisen as a response to this growing demand. DSLs are specialized programming languages designed to address specific problems in particular domains [1,2] and have become an area of research in software engineering [1–6]. By providing a high level of abstraction, DSLs allow modeling, specifying, and defining structures, specifications, and functionalities that solve domain-specific problems. The main goal of a DSL is to simplify the implementation process of a system or part of it, allowing domain experts to participate in the development of reliable, robust, and high-quality systems to address specific problems [7].

DSLs can be classified into Textual DSLs and Visual DSLs, according to [1,7,8]. In turn, Textual DSLs can be further categorized as follows: (i) DSLs with open text grammar focused on the domain context, (ii) DSLs with XML grammar (DSL-XML) based on the XML standard [9], and finally, (iii) DSLs with JSON grammar, under the JSON standard (JSON-LD, JSON for Linked Data) [10,11].

Within this last classification, RhoArchitecture [12] allows designing, implementing, deploying, and running DSL solutions that use JSON grammar (JSON-DSL) for web applications on both the client and the server side. However, although the RhoArchitecture was rigorously evaluated [12], no validation was conducted on the solutions implemented and deployed with this approach. This validation is essential to ensure that the solutions are practical and beneficial in real-world applications.

Regarding the quality of DSLs, different quality features can be assessed, such as expressiveness, flexibility, productivity, reusability, reliability, learnability, ease of maintenance, or usability [13]. This last feature, the usability of DSLs, is an important consideration for their adoption by domain experts [14], defined as the *"extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"* [15]. The usability of DSLs is related to some of the advantages associated with their adoption, such as the abstraction of the technical details of the system [16], the ease of understanding [17], or the reduction in code writing. Hoffmann et al. [18] performed an empirical evaluation of the improved effectiveness and efficiency of using DSLs versus using general purpose languages (GPLs). Their goal was to determine whether using a DSL is more effective than writing all the code and whether the results were of comparable quality.

Thus, the objective of the present article is to detail the evaluation performed on RhoArchitecture through a specific case study we called RestRho, a domain-specific solution that facilitates the specification, implementation, and deployment of RESTful (Representational State Transfer) APIs on a NodeJS [19,20]. We performed the evaluation from two main points of view. Firstly, a software metrics-based validation of the implementation provided an objective assessment of the software quality. This included evaluating complexity, difficulty, effort, estimated errors, and maintainability [21]. Secondly, an empirical validation evaluated the effectiveness, efficiency, and satisfaction of the two JSON-DSLs, which involved considering how well the DSL captures the essential concepts and operations of the problem domain, how efficiently users can complete tasks, and the impact on domain experts' usability and satisfaction within the development process [22–26].

Consequently, this work presents significant contributions in web application development, focusing on the application of JSON-based DSLs. In particular, the main contributions of this paper are highlighted below:

Specification, Implementation, and Development of RestRho: A case study of JSON-DSLs for the RhoArchitecture demonstrate how JSON-DSLs can be used to express solutions within the RhoArchitecture. RestRho is a JavaScript component that defines and implements a RESTful NodeJS server using two JSON-DSLs and template engines. We show how RestRho can simplify the specification and deployment of web services and operations.

Code Validation: A rigorous validation of the generated RestRho code is conducted using recognized metrics that offer an objective assessment of the software quality. The application of these metrics provides quantitative insights into complexity, difficulty, effort, estimated errors, maintainability, and other factors associated with RestRho, establishing a robust framework for determining the quality of software development.

Experiment with Computer Engineering Students: A controlled experiment is conducted with final-year computer engineering students to assess the effectiveness, effi-

ciency, and satisfaction associated with the use of RestRho. The quantitative and qualitative analysis of results provides a detailed understanding of the user experience and highlights perceived challenges and benefits when using RestRho compared with conventional approaches.

Analysis of Qualitative Contributions: An in-depth analysis of qualitative contributions is carried out through a usability study involving a detailed questionnaire. This analysis reveals critical participant perceptions regarding the ease of use, validity, and utility of RestRho, providing valuable insights into the user experience. The qualitative feedback emphasized quick development times and ease of use once the initial learning phase was overcome.

Identification of Challenges and Improvement Opportunities: Through the analysis of obtained results, the challenges and improvement opportunities for RestRho are identified. This contributes to a more comprehensive understanding of areas that can benefit from future developments and optimizations, enabling continuous evolution of the tool.

Collectively, these contributions aim to advance the understanding of innovative tools in web application development, providing a faster and more efficient approach that can be leveraged by a broad range of professionals, from software engineers to domain-specific experts.

Thus, the remainder of the article is structured as follows. Section 2 describes the RestRho use case for creating RESTful APIs based on RhoArchitecture. Section 3 details the validation process conducted for our use case. Section 4 presents discussions on both the code and empirical validation. Section 5 discusses threats to validity. Finally, Section 6 presents our conclusions and future work.

2. RestRho: Building REST APIs Based on JSON-DSL

This section presents the analysis, design, and implementation of RestRho, a RESTful API server based on JSON-DSL. First, it shows the details of the most relevant features that RestRho must consider; then, it outlines the RhoArchitecture and details the design and implementation of the two languages that compose the proposed solution; finally, it provides a sample of how to use RestRho.

2.1. A Look at DSL-Based Solutions to Specify RESTful APIs

The use of DSLs to specify and define API RESTful is growing due to the advantages they offer in comparison with traditional approaches for developing code [27]. These advantages include higher productivity, greater accuracy, and better documentation.

RAML RESTful API Modeling Language, Salesforce, Inc. (OpenAI, The Linux Foundation®, https://raml.org/ (accessed on 2 June 2025)) is a modeling language for describing REST API services using the YAML and JSON standards. RAML is open source and was created under the design-first development paradigm [28]. Multiple specifications are required for different tools, including development and quality control. It has tools for online modeling, but support is poor for newer versions. Additionally, code reuse and extension are limited in this model [28].

Swagger (SmartBear Software, https://swagger.io/ (accessed on 4 June 2025)) was created as a specification and a set of tools for describing, producing, consuming, and visualizing RESTful web services. In 2015, it was renamed as the OpenAPI (OpenAI, The Linux Foundation®, https://www.openapis.org/ (accessed on 15 June 2025)) specification [29], essentially the evolution of Swagger that supports both JSON and YAML formats. Like RAML, OpenAPI requires separate specifications for various tools such as development and quality control tools, and it does not support code reuse, inclusion, or extension.

WADL (Web Application Description Language, Sun Microsystems, https://www.w3.org/submissions/wadl/ (accessed on 12 February 2025)) is an XML description language readable by web application engines over HTTP, such as REST web services. Creating descriptions with WADL is time-consuming, and the binding methodology is more difficult than RAML, Swagger, and OpenAPI [28]. (API BluePrint API Blueprint, https://apiblueprint.org/ (accessed on 14 February 2025)) is another YAML-based language that focuses on the documentation aspect of APIs. It is a set of semantic assumptions that is laid on top of the Markdown syntax used to describe a web API. In addition to the mentioned tools, there are other alternatives, including Slate, GraphQL SDL, Protocol Buffers, Kotlin DSL, API Transformers, Restlet Studio, Apiary, Stoplight studio, Apigee Edge API Designer, Mulesoft Anypoint Platform, Oracle API Platform, and 3scale API Designer.

Table 1 compares the four DSLs most used for specifying and defining RESTful API, namely: RAML, Swagger/OpenAPI, WADL, and API Blueprint [30]. All tools allow the definition and/or specification of DSLs and built-in documentation. However, OpenAPI's documentation goes beyond what is provided on its website due to its widespread use. In terms of tool availability, all of them provide an Online IDE for DSL definition; however, only RAML and Swagger/OpenAPI support online API modeling. All tools provide extension and customization facilities as well as integration with external tools. None of these tools offer the use of a template engine natively for the transformation of operation results.

Table 1. Comparative analysis of tools for building RESTful APIs.

	Language Type	Web IDE	Integrate Documentation	Reuse and Modularity	Format	Validation and Code Generation	Tests and Simulations	Online Modeling	Extensions and Customization	Use Template Engines	Integration with Tools
RAML	DSL	Yes	Yes	Lim.	YAML	Yes	Yes	Yes	Yes	No	Yes
Swagger/ OpenAPI	Spec.	Yes	Yes	No	YAML, JSON	Yes	Lim.	Yes	Yes	No	Yes
WADL	DSL	Yes	Yes	Lim.	XML	Yes	Yes	No	Yes	No	Yes
API Blueprint	DSL	Yes	Yes	Lim.	YAML	No	Lim.	No	Yes	No	Yes

Note: **Spec.**: Specification. **Lim.**: Limited. **Int.**: Interpreted. **Web IDE**: Integrated web development environment.

It is worth noting that most tools are based on textual or XML DSLs. OpenAPI is the only tool that can use JSON grammar for defining its RESTful API operations. Our case study, RestRho, is based on two JSON-DSLs with reusability and modularity capabilities, facilitating the transformation of JSON operation results and allowing the use of template engines natively (as part of the language). This differentiating feature adds versatility and enhances the customization and adaptation of RESTful API services, fostering greater creativity and efficiency in the development of web applications. Part of the validation will focus on evaluating our differentiating feature.

2.2. Summary of RhoArchitecture

Since the foundation for creating RestRho is RhoArchitecture [12], Figure 1 (adapted from [12]) provides a graphical summary of the architecture for building and executing JSON-DSLs. The architecture includes several key components that work together to facilitate the development and deployment of these languages. These components ensure a streamlined process for defining, implementing, and running JSON-DSLs, thereby en-

hancing productivity and robustness in software development. The architecture includes the following:

- **Rho Programming Model (RhoModel)**: This model establishes a programming framework to add JavaScript functionality and support the corresponding code and documentation generation. RhoModel allows integrating (i) JSON-DSL specification, a programming language using JSON grammar, where RhoLanguage is a JSON-DSL combined with JavaScript classes to implement grammar elements (lower-central part of Figure 1). JavaScript components and web components [29,31] that interact with the JSON-DSL (bottom part of Figure 1). JavaScript template engines, like the Handlebars Engine, enabling the creation of custom templates efficiently (left-hand part of Figure 1). Definitions for connecting to various Data Sources such as JSON, XML, and text (top part of Figure 1).
- **Rho JSON-DSL Evaluation Engine (RhoEngine)**: This JavaScript component (top-central part of Figure 1) can run multiple programs (RhoCode) written in different JSON-DSLs. Executing a JSON-DSL program involves evaluating the functionality of the nested grammar symbols, starting from the root and drilling down based on the grammar definition. JSON-DSLs running in RhoEngine can connect and exchange heterogeneous information, use template engines, incorporate components and web components, apply security policies, and follow good programming practices [32,33], enhancing their functionality, reliability, and robustness.
- **Web-Integrated Development Environment for Rho (WebIDERho)**: This is a web-development environment that uses the RhoModel for programming and the RhoEngine for execution. WebIDERho allows us to define projects for both server and client sides, visualize class diagrams, automatically generate documentation, and deploy NodeJS-based web applications.

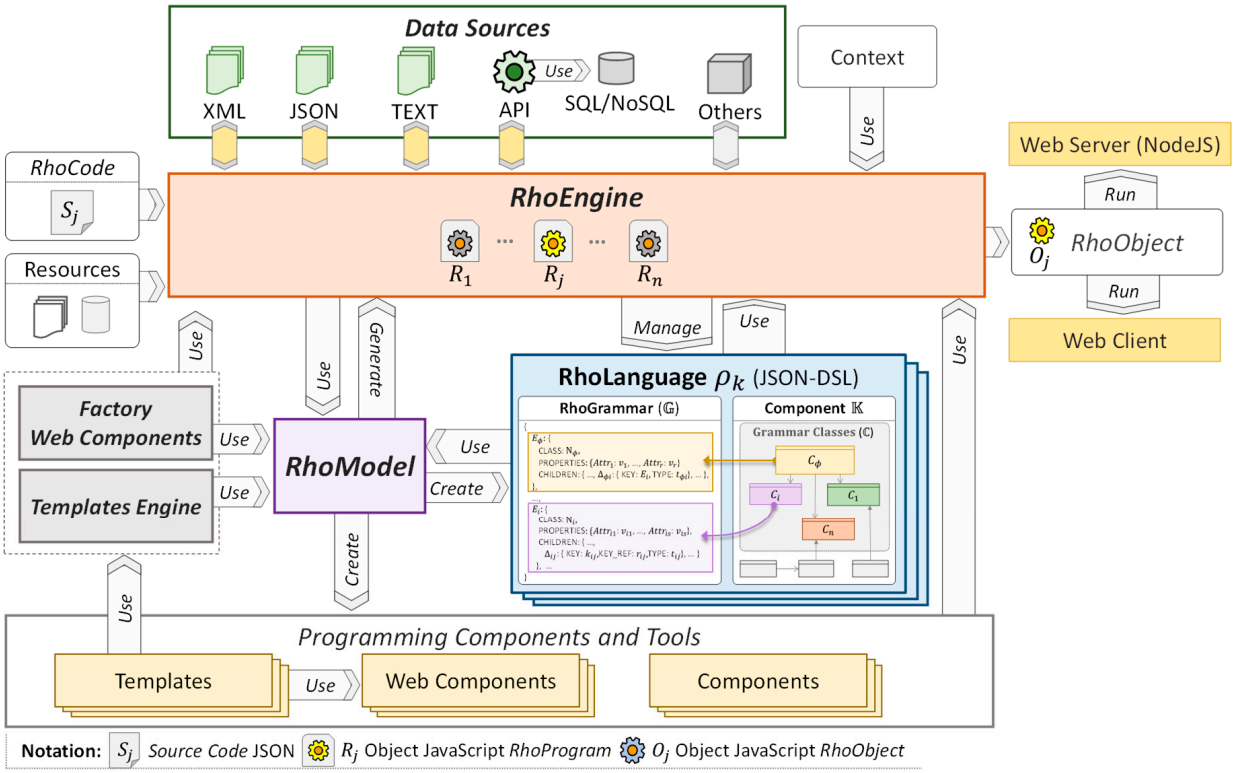


Figure 1. Summary diagram of functional blocks for the RhoArchitecture.

Using the template engines, the JavaScript components, and the web components when creating JSON-DSL, together with the Data Sources binding, exponentially increases the power and versatility of these languages. More details about RhoEngine, RhoLanguages, and RhoModel can be found in the "Rho API" submenu of http://www.devrho.com.

In [12], the RhoLanguages are formalized. The following is a summary of this formalization along with its main results: RhoEngine manages a set of RhoLanguages defined as $P = \{\rho_1, \ldots, \rho_k, \ldots, \rho_m\}$; each language $\rho_k \in P$ (see RhoLanguage ρ_k in Figure 1) is registered with an alias and defined with the tuple defined as:

$$\rho_k = \langle \mathbb{G} | \mathbb{K} | \mathbb{E} \leftrightarrow \mathbb{C} \rangle \tag{1}$$

where RhoGrammar $\mathbb{G}$ is the grammar of language ρ_k, $\mathbb{K}$ is the reusable JavaScript component that implements the functionality of the language ρ_k, $\mathbb{C}$ is the subset of classes that define the grammar's functionality for $\mathbb{G}$, and finally, $\mathbb{E} \leftrightarrow \mathbb{C}$ is the functionality that is added to each element of the language when it is being interpreted. In short, it is the implementation of a JSON-DSL named ρ_k in a JavaScript Component $\mathbb{K}$.

Since RhoEngine (see [34]) allows executing multiple RhoCode S_j programs from different languages, P, where $\mathbb{R} = \{R_1, \ldots, R_j, \ldots, R_n\}$, represents the execution instances R_j of the programs S_j that are written in a language ρ_k. In other words, it is the execution of multiple programs from different JSON-DSLs. In particular, we can define a web application W_{APP} as a subset of the execution instances of $\mathbb{R}$, and it can be expressed with the following equation:

$$W_{APP} = \left\{ S_{jk} \middle| 1 \leq j \leq n,\ S_{jk}\ coded\ in\ \rho_k \in P, 1 \leq k \leq m \right\} \tag{2}$$

and the execution of a W_{APP}, either at client or server side, can be expressed as the following set of executions:

$$Exec(W_{APP}) = \left\{ O_{jk} \middle| 1 \leq j \leq r,\ O_{jk}\ object\ reference\ of\ R_{jk} \in \mathbb{R}, 1 \leq k \leq m \right\}. \tag{3}$$

The execution of an application W_{APP} can coexist with the executions of other W_{APP} written in different $\rho_k \in P$, i.e., different JSON-DSLs. For more details on the formalization of RhoArchitecture, refer to [12].

2.3. RestRho for the Specification, Implementation, and Deployment of RESTful APIs

As previously stated, it is necessary to design, implement, and deploy a DSL to assess the RhoArchitecture, and this is where RestRho comes in. Particularly, RestRho is a JavaScript component that aims to define and dynamically deploy HTTP requests on a RESTful NodeJS server called ServerRestRho (Source code: https://www.devrho.com/restrho/js/RestRho2.js (accessed on 10 June 2025) (see Figure 2, right side).

To achieve this, RestRho (Source code: https://www.devrho.com/restrho/js/ServerRestRho.js (accessed on 10 June 2025)) defines and deploys two RhoLanguages (see Figure 2, middle part), called SQLRho to define how to access and operate the databases and DBRestRho to define how to specify the HTTP requests and manage them by indicating the corresponding SQL template and associated SQLRho:

DBRestRho enables two main tasks: (i) to define the list of HTTP requests and associate the corresponding template to an SQL statement for each operation and (ii) to execute SQLRho code, which implies the possibility of multiple databases on the same RESTful API. DBRestRho has as an associated subcomponent DBRest of RestRho (see Figure 2, bot-

tom). Formally, according to Equation (1), the formal definition of DBRestRho is given by the following:

$$DBRestRho = \langle \mathbb{G}_{DBRest} | DBRest | \mathbb{E}_{DBRest} \leftrightarrow \mathbb{C}_{DBRest} \rangle.$$

SQLRho allows configuring a DB connection (in this case, of study MariaDB or MySQL) and defines a set of operations on existing tables. There are two kinds of operation associated with a table: (i) basic operations at record level (Insert, Update, Delete, and Select), and at table level (Where and SelectAll); and (ii) custom operations that support configuration and parameters in their definition as well as transformations and templates applications for the results of the queries. SQLRho has an associated SQL subcomponent of RestRho (see Figure 2, bottom). Formally, according to Equation (1), the formal definition of SQLRho is given by the following:

$$SQLRho = \langle \mathbb{G}_{SQL} | SQL | \mathbb{E}_{SQl} \leftrightarrow \mathbb{C}_{SQL} \rangle.$$

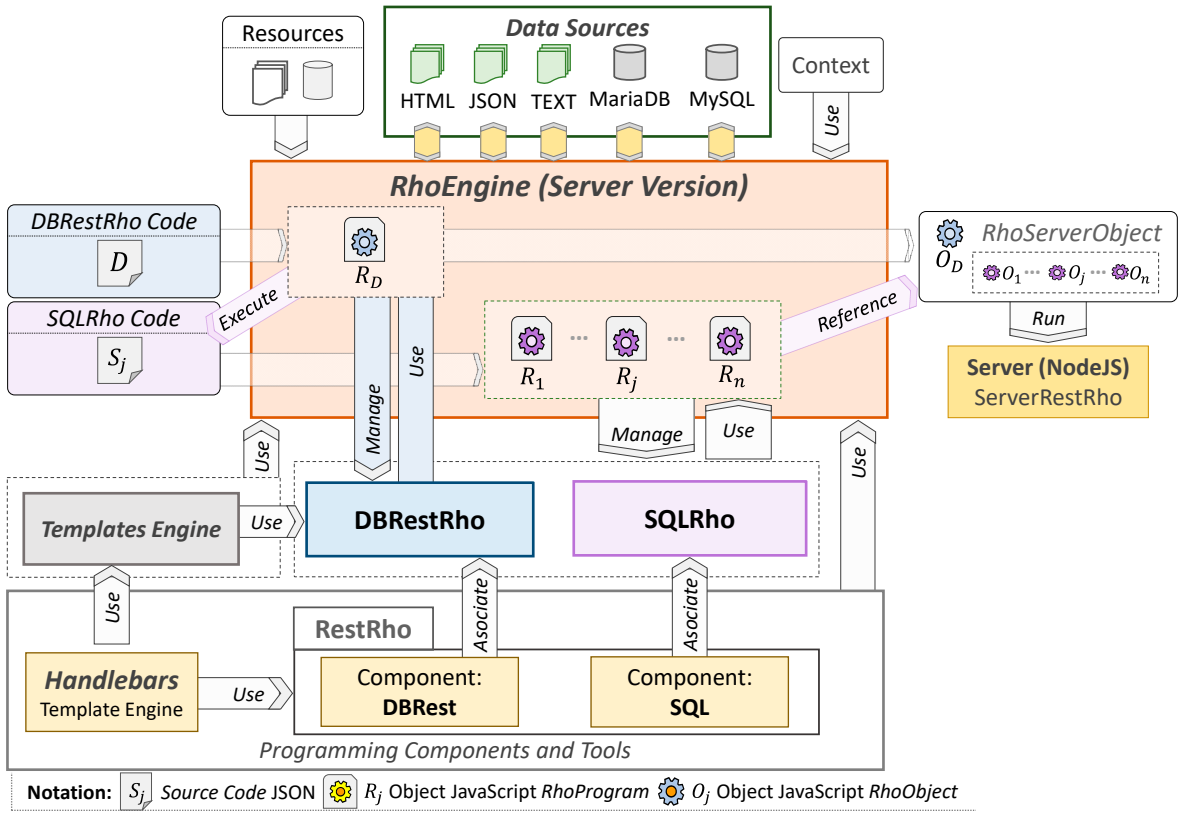


Figure 2. Diagram of functional blocks for RestRho and ServerRestRho (RESTful API server based on JSON-DSL).

The template engine used is the *Handlebars Engine* (see Figure 2, bottom left), a tool that separates the logic and the design when generating web content [11,12] (HTML code, JavaScript, SVG, CSS3, etc.). This template engine is used by both languages.

Overall, it works as follows: (i) The ServerRestRho is executed on a port. Then, (ii) the RestRho component is initialized with the execution of the DBRestRho Code program D, that interprets, creates, and makes available the request definition list, creating a RhoServerObject O_D. Later on, (iii) gets the set of SQLRho Code programs S_j for $1 \leq j \leq n$ and executes them, creating the set of programs $\mathbb{R} = \{R_1, \ldots, R_j, \ldots, R_n\}$, and its resulting set of objects $\mathbb{O} = \{O_1, \ldots, O_j, \ldots, O_n\}$ is referenced within RhoServerObject O_D. Lastly, (iv) the execution of an S_j consists of creating the database connection (MariaDB

or MySQL); compiling the *Handlebars* templates and making them available (increases response speed); and creating and publishing the basic and custom operations using the definitions from O_D.

Starting from Equation (2), the approach ServerRestRho W_{SERVER} can be expressed as follows:

$$W_{SERVER} = \{D, S_j | D \in \text{DBRestRho, and, } S_j \in \text{SQLRho}, 1 \leq j \leq n\},$$

and the execution of ServerRestRho W_{APP}, according to (3), can be expressed as follows:

$$\text{Exec}(W_{SERVER}) = \left\{O_D,\ O_j \middle| O_D \text{ object reference } R_D, O_j \text{ object reference of } R_j \in \mathbb{R}, 1 \leq j \leq n\right\}.$$

To facilitate the understanding of DBRestRho and SQLRho, the next two subsections show the JSON structure of a program with a brief description of the grammar of these two languages, respectively. Finally, the last subsection summarizes the implementation of RestRho.

2.3.1. DBRestRho Language

Algorithm 1 shows a brief explanation of a DBRestRho program. It has four attributes: Name (where <name> is the name for the Server/Program), Description (where <description> is a brief description for the list of definitions), Databases, and Definitions.

The Databases attribute is an array of SQLRho programs, where each <SQLRhoCodeI> is the name of the source file for the SQLRho program, and can be seen as:

<SQLRhoCodeI> = "<alias>.<connector>.json",

where <alias> is the program alias and <connector> is the connector to the corresponding database, either "mariadb" or "mysql".

The attribute Definitions is a map of HTTP requests, and each request is associated with an SQL statement (Handlebars template with an SQL statement). The <definition> is the name of the operation (for example, Insert, Update, Delete, Select, or SelectAll), and five attributes are established:

(i) **Method**: is the method of the HTTP request, where <method> is the name of the method to use: get (retrieves resources), post (creates new resources), put (updates resources), and delete (remove resources).

(ii) **Parameters**: is the list of parameters (<param1>, …, <paramN>) to use within the HTTP request and SQL statement. For example, {"params": ["key"]}, and its use in the template is ":key".

(iii) **Default**: are the default options (<option1>, …) to be used in an SQLRho program. For example, {"Fields": "*"}, means that it takes the field specification of a table and lists the names separated by commas ",".

(iv) **Path**: is the URL structure of the HTTP request (Handlebars template). Generally, <templatePath> can be expressed as follows:

$$< \text{templatePath} >= \underbrace{\text{"/\{\{db\}\}/\{\{table\}\}}}_{\text{Path}} \underbrace{[/\{\{\text{action}\}\} | / < \text{resource} > | / < \text{params} >]\text{"}}_{\text{Resource}} \quad (4)$$

where /{{db}}/{{table}} is the path of the URL, with the {{db}} name defined in the SQLRho program, and {{table}} is the name of the table to use. The Resource of the URL can be defined in three ways:

1. /{{action}}: an action that is defined in a table of an SQLRho program.
2. /<resource>: a fixed resource. For example, <resource>=/all.

3. /<params>: the list of parameters defined in Params. For example, <params>=/:key.

(v) SQL is the SQL template specified in Handlebars (Minimal templating on steroids: https://handlebarsjs.com/ (accessed on 12 July 2025)) where there is a predefined set of tags required for an SQL statement: Table, Fields, Distinct, Join, InnerJoin, LeftJoin, RightJoin, FullJoin, Where, GroupBy, Having, OrderBy, Limit, are added to Parameters. For example:

<templateSql> = "UPDATE {{Table}} SET {{{Fields}}} WHERE {{key}}=:key"

where {{Table}} is the name of a table, {{{Fields}}} is defined in Default, and {{key}} is a parameter. It should be noted that "{{{" is used to escape special characters in Handlebars.

Algorithm 1 RhoCode DBRestRho

```
{
 "Name": "<name>",
 "Description": "<description>",
 "Databases": ["<SQLRhoCode1>", ..., "<SQLRhoCodeI>", ...,
"<SQLRhoCodeN>"],

 "Definitions": {
      ...
     "<definition>": {
       "Method": "<method>", // <method>: get|post|put|delete
       "Parameters": {"params": ["<param1>", ..., "<paramM>"]},
       "Default": {"<option1>": "*", ...},
       "Path": "<templatePath>",
       "SQL": "<templateSql>"
   },
   ...
  }
}
```

2.3.2. SQLRho Language

In the Algorithm 2, a summary explanation of an SQLRho program is shown. It has eight defined attributes: Name (where <db> is part of the Path, which is defined in (1) for the HTTP request), Description (where <description> is a brief description of the program), Connector (the connector type database, <conector>=[mariadb|mysql]), EnableToken (enables the use of the token for any HTTP request, <enable>=[true|false]), Pool (defines the standard connection pool to a MariaDB (https://mariadb.com/docs/connectors/mariadb-connector-nodejs/getting-started-with-the-node-js-connector (accessed on 10 June 2025)) database or MySQL (https://www.npmjs.com/package/mysql/ (accessed on 10 June 2025)), Registry, Templates, and Tables.

Algorithm 2 RhoCode SQLRho

```
{
  "Name": "<do>",
  "Description": "<description>",
  "Connector": "<connector>", // <connector>: mariadb|mysql
  "EnableToken": <enable>, // <enable>: false|true
  "Pool": {"host":"…", "user":"…", "password":"…", "database":"…", "connectionLimit":<int> },
  "Registry": {"Table":"…", "User":"…", "Password":"…", "ValidityMinutes": <int> },

  "Templates": { ..., "<nameTemplateI>": {"Dir":"<dirI>", "File":"<nameFileI>"}, ... },

  "Tables": {
     ...,
     "<tableI>": {
       "Encrypt": ["<field1>", ..., "<fieldK>", ...],
       "Decrypt": ["<field1>", ..., "<fieldJ>", ...],
       "Uses": ["use1", ..., "useI", ..., "useN"],
       "Operations": {
         ...
         "<operationI>": {
           "Name": "<nameOperation>",
           "Action": "<action>",
           "Parameters": {"query": ["<param1>", ..., "<paramJ>"] },
           "Clauses": {"Fields": "…", "Where": "…", "OrderBy": "…", ... },
           "Transform": {"Type": "<type>", ..., "<varI>": ... },
           "Template": "<nameTemplateI>"
         }
       }
     },
     ...
   }
}
```

The attribute Registry is an object defining the database record table. The attribute Table is the name of the user table, the attribute User is the name of the user's field, the attribute Password is the name of the field password, and the attribute ValidityMinutes is the time in seconds of the validity of the generated token. When EnableToken is enabled, the operation of the login to obtain the token has the following structure:

<protocol>://<host>/<db>/login?User=<user>&Password=<password>

where <protocol> (http|https) is the configured protocol, <host> is the host and port of the service, login is the resource, <user> and <password> are the input parameters. This token must be sent in the protocol header in all the operations to be performed.

The attribute Templates is an object defining the set of Handlebars templates to be used for transformations of the results of table operations, where <nameTemplateI> is the template name, <dirI> is the directory, and <nameFileI> is the filename of Handlebars.

Finally, the attribute Tables is the set of Tables to which HTTP requests shall target. For each <tableI> (database table name), four attributes are set:

(i) **Encrypt**: is the list of field names <FieldK>, of the table <tableI> to be encrypted for sending. For data encryption, RestRho uses an AES-256 algorithm in CBC (Cipher Block Chaining) mode. Sensitive fields are usually encrypted to protect confidential information (table identifiers, passwords, personal data, medical history, bank details, etc.).

(ii) **Decrypt**: is the list of field names <FieldJ> of the table <tableI> to be decrypted on the server for processing. Similarly, RestRho uses the same AES-256 algorithm in CBC mode to decrypt the data.

(iii) **Uses**: element defines the set of fundamental operations available for <tableI>. These include record-level actions—**Insert**, **Update**, **Delete**, and **Select**—as well as table-level operations such as **Where** clauses and the **SelectAll** operation.

(iv)) **Operations**: is the list of custom operations that support configuration and parameters in their definition, and transformations and templates can be applied to the results.

2.3.3. Implementation Summary

Figure 3a shows the RestRho component diagram. The RHO component of RhoArchitecture is explained at [12]. The DBRest subcomponent implements the functionality associated with the DBRestRho language, while the SQL subcomponent implements the functionality associated with the SQLRho language.

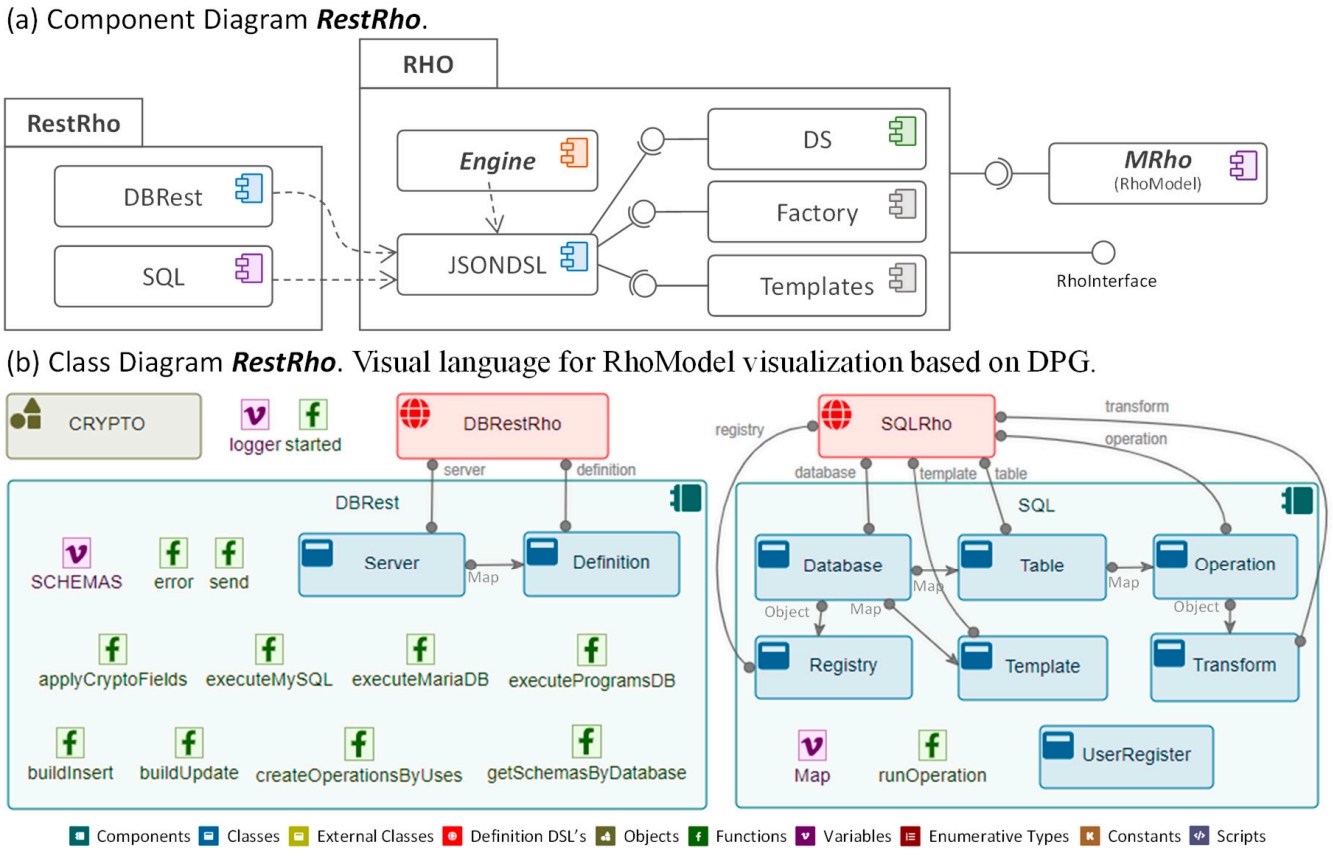


Figure 3. The top part of the figure shows the Component Diagram of the RestRho, and the bottom part shows the Class Diagram SQLRho. Component Diagram RestRho and Class Diagram SQLRho.

For its part, Figure 3b shows the RestRho class diagram. This diagram was created with Diagram Programming Generate DPG [34] and is a JSON-DSL that allows you to specify programmable diagrams based on PsiDiagram [35] for web applications. Detailed help on the RestRho class diagram is generated by WebIDERho and can be found at http://www.devrho.com?doc=restrho (accessed on 10 June 2025).

The SQLRho language is found in Figure 3b on the right. It shows the elements of the grammar,

$$\mathbb{E}_{SQL} = \{\text{“database”}, \text{“registry”}, \text{“template”}, \text{“table”}, \text{“operation”}, \text{“transform”}\},$$

with their respective associated classes and their root element $E_\phi =$ “database”. It should be noted that the tree of nested elements and their type is displayed, e.g., E_ϕ has defined the CHILDREN: “Registry” (type Object), “Templates” (type Map), and “Tables” (type Map).

The DBRestRho language is shown in Figure 3b on the left. It shows the elements of the grammar $\mathbb{E}_{DBRest} = \{\text{“server”}, \text{“definition”}\}$ with their respective associated classes and their root element $E_\phi =$ “server”.

2.4. Sample RestRho

The purpose of this section is to create an illustrative example to show the implementation of the DBRestRho and SQLRho programs and their running on ServerRestRho.

Figure 4 shows the Sample RestRho data model diagram. This model is implemented in MariaDB Server Community version 11.1.2-GA. The name of the database is SampleRestRho, and it is made up of four tables: user, person, position, and employee.

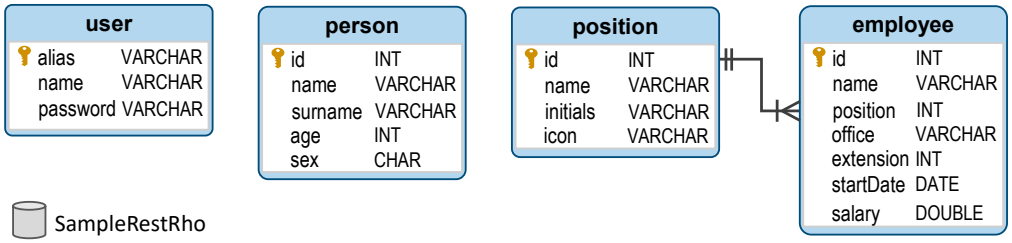


Figure 4. Data model diagram sample RestRho.

Figure 5 shows the program DBRestRho “rho.json” (on the left), denoted by D. In its Databases list, the “sample.mariadb.json” SQLRho program is defined (on the right, connected to the database SampleRestRho), denoted by S_1. In addition, in S_1, the record table is *user*.

Figure 5 also shows didactically how the HTTP **Path** request is generated and the **SQL** sentence of the Update operation for table person. For the **Path**, RestRho dynamically creates the PUT request.

On the other hand, in the **SQL** statement, in addition to the table named “person”, the values of each field are the values that come in the Query String, and the primary key (id field) is the key parameter.

Another interesting example to address is shown in Figure 6, with a customized operation SelectPillTemplate for table *person*. Figure 6a shows once again the programs *D* (extract of the definition of SelectBasicAction) and S_1 (extract of the operation SelectPillTemplate). The Handlebars template “pills.html” is used in this operation (see Figure 6a) based on the Bootstrap framework v.5.3 (https://getbootstrap.com/); it goes through all the records and for each one, it creates a colored Pill according to the age range and an icon if male or female. The result is displayed in Figure 6c.

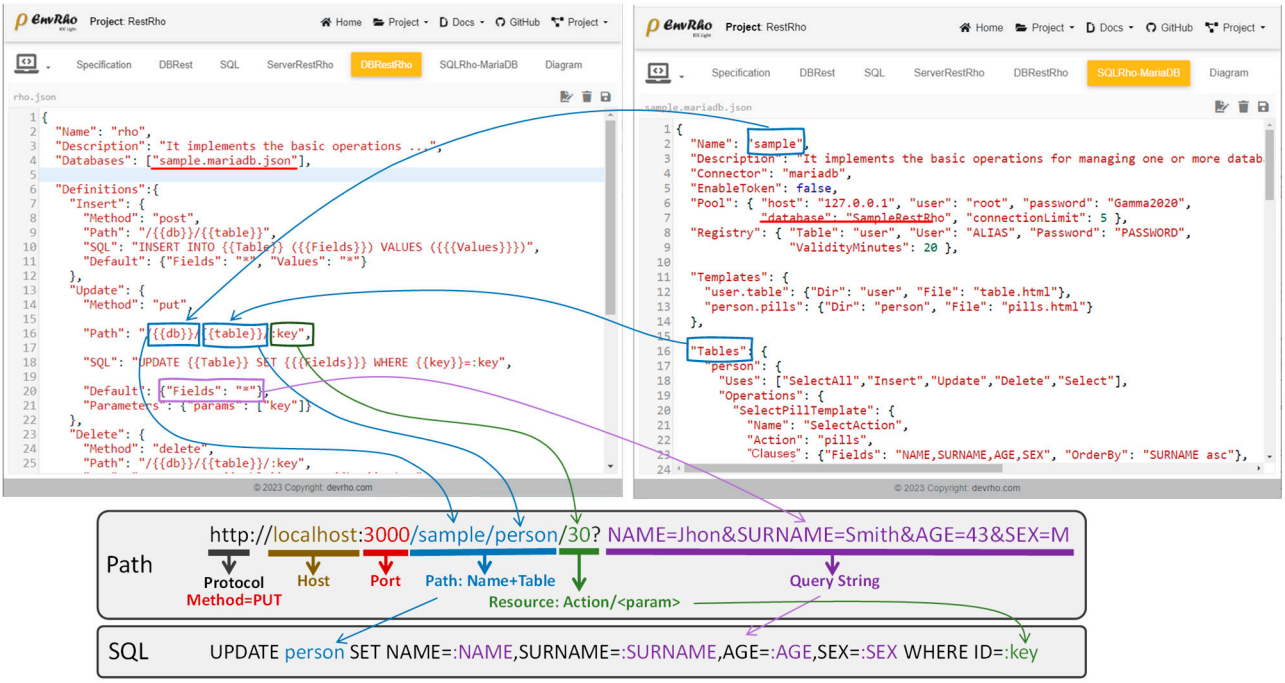


Figure 5. Programs "rho.json" (DBRestRho) and "sample.mariadb.json" (SQLRho) and how the HTTP **Path** request and the SQL statement are created.

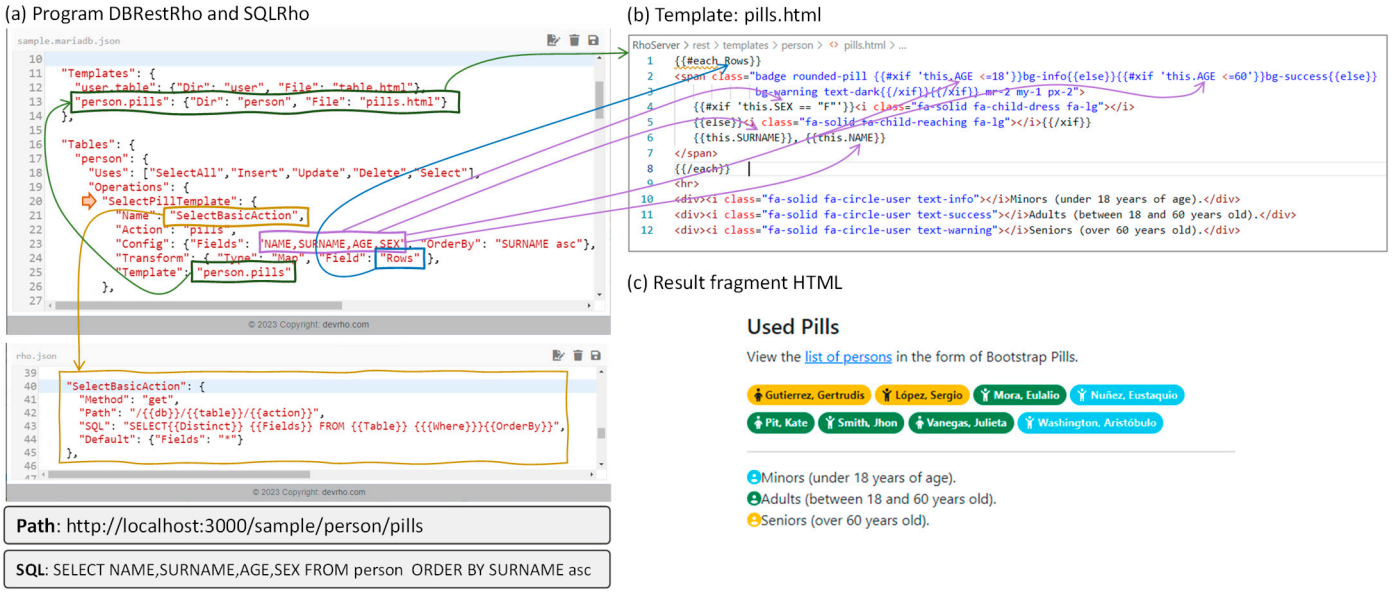


Figure 6. Programs "rho.json" (DBRestRho) and "sample.mariadb.json" (SQLRho) and how to create a transformation and a template. (**a**) Program DBRestRho and SQLRho; (**b**) File template "pills.html"; (**c**) Execution result (HTML fragment).

The running example of Sample RestRho is shown in Figure 7 and is available at http://www.devrho.com/restrho/sample.html (accessed on 10 June 2025). An example of Login and how to obtain the token is shown. From the *user* table, a table with the list of users is shown. From the *person* table, the CRUD administration of persons, the list of persons in Pills, and the percentage graph of men and women are shown. Finally, the *employee* table shows the employees in the form of Cards.

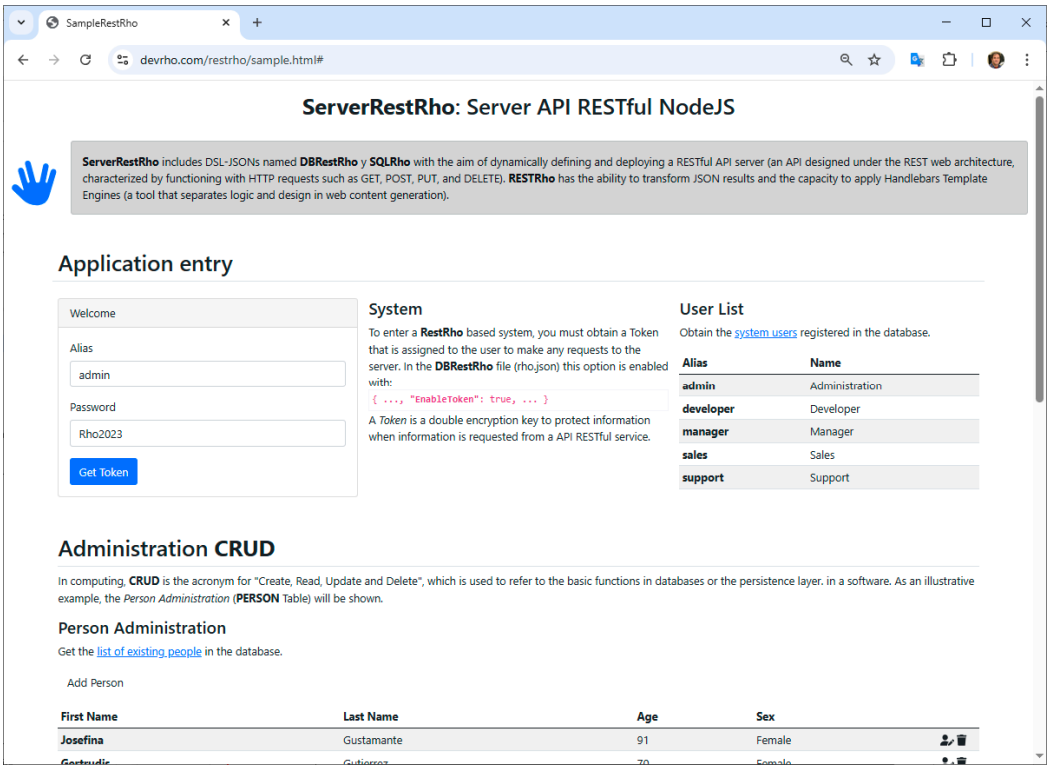


Figure 7. An illustrative example of RestRho: Sample RestRho.

2.5. Final Comments

Throughout this section, we have presented the formalization and implementation of two languages, namely, DBRestRho and SQLRho. These are JSON-DSL based on RhoArchitecture for the creation of a RestRho component that aims to dynamically define and deploy HTTP requests on a RESTful NodeJS API server called ServerRestRho. The RestRho component was specified and implemented with the WebIDERho integrated web development environment. Finally, an illustrative example (*SampleRestRho*) was created to show the implementation of DBRestRho and SQLRho programs and their operation on our RestRho.

3. Validation and Experimentation

This section aims to show the validation and experimentation carried out to assess the RhoArchitecture using the designed languages. On the one hand, it evaluates the DSL software by measuring its quality with well-known metrics. On the other hand, it details the quantitative and qualitative evaluation performed by users.

3.1. Code Validation

In the field of software engineering, software metrics are quantitative tools that provide an objective assessment of various characteristics of an information system. Several systematic reviews [36–38] focus on key aspects such as quality, reliability, documentation, and software complexity. In the context of our proposal, software metrics should focus on the analysis of JavaScript Components and should be adjusted for prototype-based languages such as JavaScript [39].

The software metrics that we have chosen to use in our analysis are as follows: (i) *Lines Codes* [40] (LOC, physical lines code SLOC, logical lines code LLOC, comment lines code CLOC, and blank lines code BLOC); (ii) *Cyclomatic Complexity* [41] (average per-function cyclomatic complexity CNN, and cyclomatic complexity density for module CND [42]);

(iii) *Halstead Metrics* [42] (Halstead vocabulary size per module HS, Halstead difficulty per module HD, Halstead volume per module HV, Halstead effort per module HE, Halstead bugs per module HB, Halstead time per module HT, and Average Halstead effort per function HEF); and (iv) *Maintainability Index* MI [43].

Table 2 shows the JavaScript code generation summary for the RhoArchitecture Components (RhoEngine, RhoModel, WebIDERho, denoted by CRA, in yellow color), our RESTful NodeJS API server (RestRho, ServerRestRho, denoted by SRR, in blue color) and the main frameworks used by SRR (Handlebars, Express, MariaDB, denoted by FRR, in purple color).

Table 2. Components of the RhoArchitecture (in yellow), ServerRestRho (in blue), and frameworks using RestRho (in violet): JavaScript generation code summary.

Project	File	RhoModel			JavaScript Generated Code					Conciseness
		MRho	MIRho	RHOLOC	LOC	SLOC	LLOC	CLOC	BLOC	RHOLOC/ SLOC
RhoEngine	Rho.js (client) Rho.js (sever)	237(1)	1578(6)	1815	2375 1598	1347 847	1350 809	714 510	314 241	2.19
RhoModel	MRho.js	277(1)	949(6)	1226	2140	1518	1055	516	106	1.75
WebIDERh	WebIDERho.js	129(1)	910(3)	1039	1232	812	886	287	133	1.19
RestRho	RestRho.js	86(1)	717(2)	803	1186	1177	782	302	118	1.47
ServerRest	SeverRestRho.js	-	-	-	21	11	11	4	3	-
Handlebar	Handlebars-all.js	-	-	-	3351	2575	1643	311	465	-
Express	Express-all.js	-	-	-	4054	1820	1379	1575	659	-
MariaDB	MariaDB-all.js	-	-	-	14,318	11,531	8434	1635	1152	-
	Total	729(4)	4118(17)	4847	30,275	21,638	16,350	5859	3189	1.65

Note: X(Y) MRho/MIRho; X: number of Rho lines, Y: number of Psi files; RHOLOC: total Rho lines of code; LOC: JavaScript-generated lines; SLOC: physical executable code lines; LLOC: logical executable code lines; CLOC: comments code lines; BLOC: blank code lines.

The RestRho component was generated using RhoModel [35,44]. Altogether, a total of 3 files were created, i.e., 1 MRho specification file (33%) and 2 MIRho implementation files (67%). The total amount of source code for RestRho was 803 lines, which generated a total of 1186 lines of JavaScript code, with an SLOC of 1177, which means a conciseness ratio of 1.47, which is in the range of the CRA components. On the other hand, in SRR, the CLOC comment lines represent 34% on average, which is a highly documented assessment rate (>25%, according to [43,45]). This highlights the quality of RestRho's documentation.

Figure 8 gives the percentage of SLOC, LLOC, CLOC, and BLOC with respect to the total number of LOC lines for all the components/frameworks analyzed in this article. From the total number of lines of code (30,638), 72% are FRR, 24% are CRA, and 4% are SRR. Furthermore, the "good documentation" of CRA and SRR is ratified between 20 and 43%. It is noteworthy that the best documented is ServerRestRho, with 43%, which is the highest score in the study. BLOC is between 10 and 20% for most components/frameworks, while SLOC is between 45 and 81%, with Express being the highest at 81%.

Table 3 provides a summary of the software metrics (cyclomatic complexity, Halstead metrics, and maintainability index) distributed over the three sets CRA, CRR, and FRR.

Overall, according to Table 3, the obtained software metrics for CRR look very good. On the one hand, the cyclomatic complexity reflects "simple functionality, without too much risk" (CNN < 5 according to [41]). On the other hand, according to [39,42], and not taking into account ServerRestRho, our RestRho component has the lowest complexity (HS = 449), the lowest associated functionality (HV = 38,414), one of the lowest difficulty levels

(HD = 97), the shortest implementation time (HT = 207,299), the lowest estimated error per module (HB = 12.8), and the lowest comprehension effort (HEF = 1439).

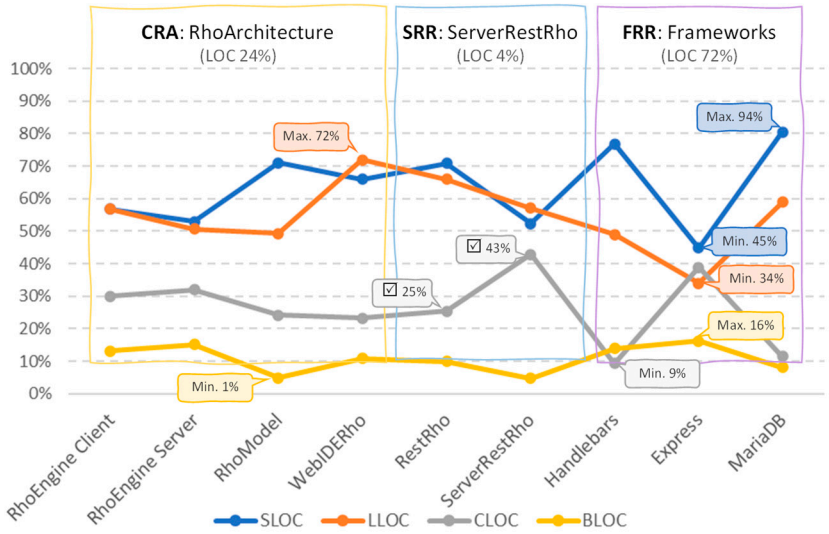


Figure 8. Comparison of the SLOC, LLOC, CLOC, and BLOC metrics between components RhoArchitecture CRA, ServerRestRho CRR, and frameworks used RestRho FRR.

Table 3. Components of the RhoArchitecture (in yellow), ServerRestRho (in blue), and frameworks using RestRho (in violet): JavaScript generation code summary.

Framework/Component	CNN	CND	HS	HD	HV *	HE $^{+}$	HEF	HB	HT *	MI
RhoEngine v1.1 (client)	1.9	14.3	610	154	68.9	10.63	2663	23.0	590.4	114.3
RhoEngine v1.1 (server)	1.8	13.7	413	140	41.2	5.78	2082	13.7	321.4	116.6
RhoModel v1.0	3.5	20.1	626	176	95.2	16.73	11,073	31.7	929.4	97.9
WebIDERho v1.1	1.7	9.4	575	118	54.8	6.45	2336	18.3	358.5	114.2
RestRho v1.0	1.4	7.9	449	97	38.4	3.73	1439	12.8	207.6	121.1
ServerRestRho v1.0	2	18.2	25	5	0.3	0.001	1525	0.1	0.1	106.9
Handlebars v5.0.0	2.5	20.8	887	134	98.4	13.3	2964	32.8	737.3	113.4
Express v4.16.4	3.0	21.5	742	148	81.0	12.0	3279	27.0	667.9	109.8
MariaDB v2.5.6	5.9	10.8	4346	96	306.4	29,6	54,784	102.1	1643.0	84.1

Note: CNN: cyclomatic complexity average per function; CND: cyclomatic complexity per module; HS: Halstead vocabulary size per module; HD: Halstead difficulty per module; HV: Halstead volume per module (* thousands); HE: Halstead effort per module ($^{+}$ millions); HEF: average Halstead effort per function; HB: Halstead bugs per module; HT: Halstead time per module (* thousands); MI: maintainability index.

In addition, according to [39,42], Figure 9 confirms the lowest average values for CRR: the lowest complexity and associated functionality (HS = 237, HV = 19.370), the lowest level of difficulty (HD = 51), the shortest implementation time (HT = 103.691), the lowest error estimate per module (HB = 6.5), and the lowest comprehension effort (HEF = 1.471).

In comparison, the HD difficulty level of CRR is about one-third the difficulty of CRA and one-half the difficulty of FRR. On the other hand, it is relevant to see that CRR's HEF comprehension effort is approximately fourteen times lower than FRR and three times lower than CRA. The difference in CRR is also noticeable in the lower estimation error per module, eight times lower than FRR and three times lower than CRA.

Another fact to highlight is that in the MariaDB framework, most of Halstead's metrics are higher than the rest of the frameworks and components.

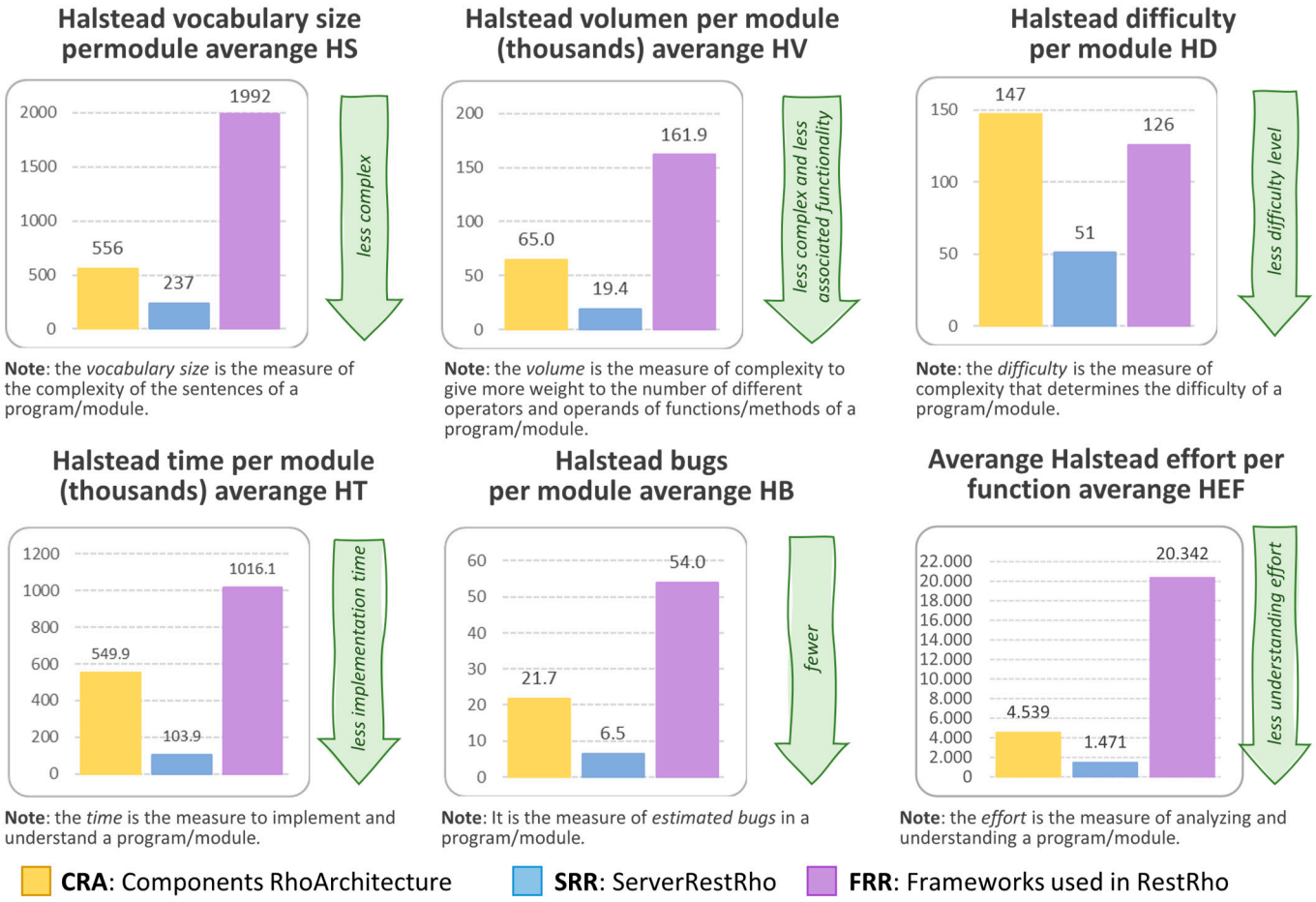


Figure 9. Graphs of the Halstead metrics HS, HV, HD, HT, HB, and HEF for the components RhoArchitecture CRA, ServerRestRho CRR, and frameworks used RestRho FRR.

Meanwhile, Figure 10 shows the graph of the software measures of cyclomatic complexity for the CRA, CRR, and FRR. For all components, it reflects that the functions/methods have a "simple functionality, without much risk" (CNN < 5 according to [41]). On the other hand, [45] shows that the lower the CND value, the simpler the productivity and maintainability of the software becomes; it is worth noting that the CND value of our main component, RestRho, is lower than all. In summary, our RestRho has better productivity, better maintainability, and less risk than any component in this study.

Finally, Figure 11 represents the maintainability index (MI) for CRA, CRR, and FRR. As can be seen, all components and frameworks have a "good maintainability" (>85, according to [43]) The CRR components, on average, are better than CRA and FRR, although the MariaDB frameworks do not have adequate maintainability.

In summary, the results of the software metrics provide strong evidence of the quality and maintainability of the RestRho component. Specifically, the low cyclomatic complexity, high documentation ratio, and favorable Halstead and maintainability index values indicate that the codebase is well structured, understandable, and maintainable. These metrics, when considered together, support the conclusion that RestRho is a lightweight and efficient solution for building RESTful APIs using JSON-DSLs, aligning with the objectives of the study.

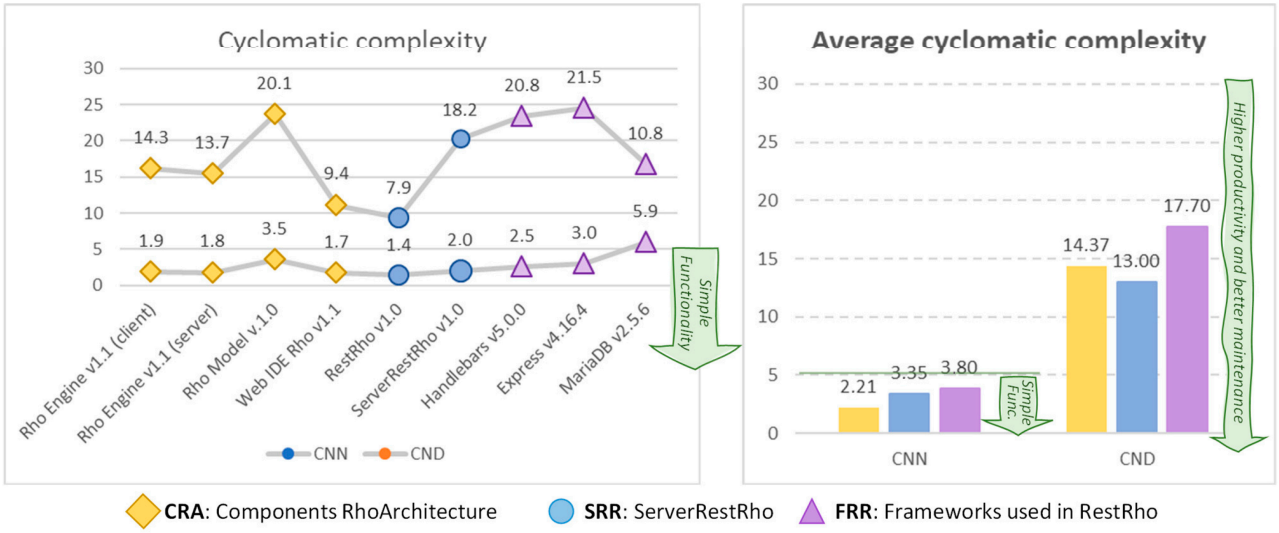


Figure 10. Cyclomatic complexity graphs for the components RhoArchitecture CRA, ServerRestRho CRR, and frameworks used RestRho FRR.

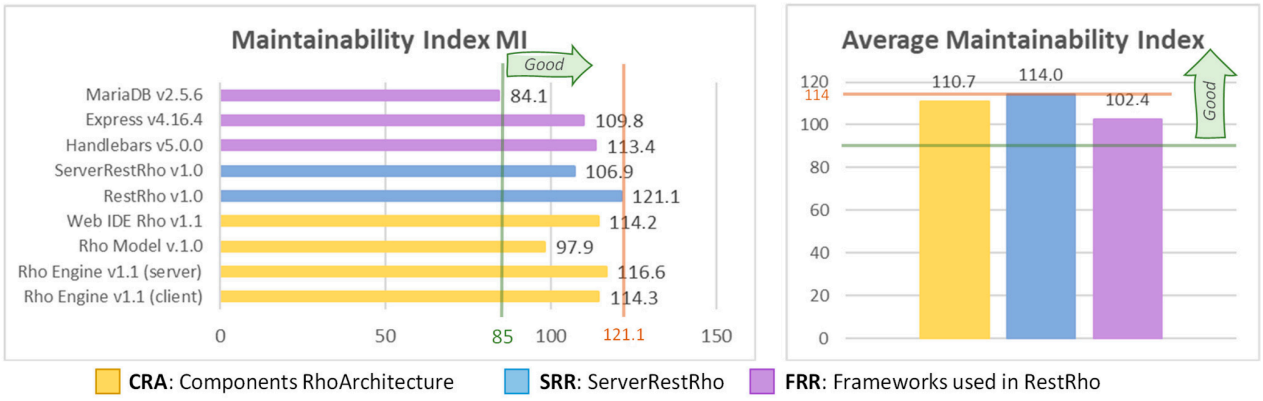


Figure 11. Maintainability Index MI for components RhoArchitecture CRA, ServerRestRho CRR, and frameworks used RestRho FRR.

3.2. Empirical Validation

This section focuses on the empirical validation of the proposal by engaging real users in a comprehensive study of the experience. This analysis was conducted from two complementary perspectives: quantitative and qualitative. Through direct interaction with the participants, the aim was to assess the usability and acceptance of the proposal. This endeavor provided an in-depth understanding of how the proposed implementation impacts and enhances the user experience in a practical, real-world setting.

3.2.1. Experimentation Setup

In this subsection, we describe the research questions stated, the participants involved, the experimental procedure defined, and the tasks they performed.

Research Question Statement

Research questions driving this RestRho case study are as follows:

- **RQ1**: What facets of user experience, including effectiveness, efficiency, and satisfaction, stand out when employing RestRho for defining and deploying a RESTful API server, as highlighted by both quantitative and qualitative analyses?
- **RQ2**: How do user perceptions of effectiveness, efficiency, and satisfaction differ when using RestRho to create a RESTful API server, considering their prior experi-

ence in programming, database design, and web development, as derived from both quantitative and qualitative analyses?

To answer these questions, we employed a mixed-methods approach. Effectiveness was measured through the success rates of task completion during the experiment, providing a quantitative assessment of how well users could accomplish predefined tasks using RestRho. Efficiency was inferred from the task completion rates and the time taken to complete each task, offering insights into the users' ability to perform tasks within a reasonable timeframe. Satisfaction was evaluated using a usability questionnaire, capturing qualitative feedback on the users' perceived ease of use, utility, and overall experience with RestRho. This combination of quantitative and qualitative data enabled a comprehensive evaluation of RestRho's performance in real-world scenarios.

Participants

The participants using RestRho were final-year students pursuing a degree in computer engineering. Written informed consent was obtained from all participants. A total of 39 senior students in the computer engineering program (Mage = 23.5 years, SD age = 1.8, ages 22 to 29; 90% male) participated in this study. The participants had experience in programming languages, database design and management, and web application development.

Experimental Procedure

The experimentation initiated with an introductory session aimed at acquainting the users with RestRho. This comprehensive session, spanning 2 h and 10 min, meticulously elucidated and demonstrated RestRho's diverse functionalities and features.

Following this introductory session, a subsequent phase commenced where participants were allocated a set of 18 tasks for the execution phase of the experiment (see the Tasks section for further details). These tasks encompassed the practical application of RestRho, requiring participants to utilize and implement RestRho to achieve the specified objectives.

Upon the completion and submission of the designated tasks, participants were invited to engage in a two-fold process. Firstly, they were invited to complete a usability questionnaire tailored to assess their experiences. This questionnaire was based mainly on the USE questionnaire [46], with some variations provided by Davis's Perception of Utility and Ease of Use questionnaire [47] as well as the Purdue Usability Questionnaire [48]. Secondly, participants were prompted to respond to specific open-ended questions aimed at eliciting in-depth insights for a qualitative study.

This sequential procedure facilitated an in-depth exploration of RestRho's functionalities, transitioning participants from an informative session to practical application. It culminated in a comprehensive assessment of user experience and usability through the administration of the usability questionnaire and qualitative inquiries for the subsequent qualitative study.

Tasks

Table 4 provides a detailed overview of the tasks assigned to users within the scope of the experiment. These tasks were crafted based on the publicly available Employees Sample Database provided by MySQL [49], offering a practical foundation for the experimental procedures. The purpose of these tasks was to define and dynamically deploy a RESTful API server using the proposed Domain-Specific Language with JSON grammar (JSON-DSL). Each task is identified by a unique code (ID) and is described clearly and succinctly in the "Tasks" column. These tasks are organized under main activities in the "Main Activity" column, providing additional context regarding the specific purpose or application of the task within the experimental environment. Furthermore, the "Task Type" column

specifies the category to which each task belongs, classifying them according to their operational nature. Finally, the "Weighting" column indicates the relative importance assigned to each task within the context of the experiment, with a total value of 1 representing the complete set of tasks. This weighting is instrumental in establishing priorities and levels of difficulty for each activity.

Table 4. Tasks overview and classification in RestRho experimentation.

ID	Tasks	Task Weighting	Task Type	Main Activity
T1	Setting up a file to launch an SQLRho program.	0.0750	TT1. Configure the SQLRho program with the 'employees' database	MA1. SQLRho program
T2	Query the current number of employees in each department, sorted alphabetically by department name.	0.0450	TT2. Operations on the 'departments' table	
T3	List all managers who have worked in a department in chronological order.	0.0450		
T4	List the current managers of each department, sorted by last name and first name.	0.0450		
T5	List the current employees in a department, including their current age.	0.0450		
T6	Query the current number of men and women in each department, sorted by the number of employees in descending order.	0.0600		
T7	Query the current minimum, maximum, and average salaries by department, sorted by average in ascending order.	0.0600		
T8	Query the current number of men and women in the company.	0.0300	TT3. Operations on the 'employees' table	
T9	Query the titles that an employee has held in chronological order.	0.0450		
T10	Query the departments where an employee has worked in chronological order.	0.0450		
T11	Query the current number of men and women in the company within the following age brackets [0–17, 18–40, 40–65, 65+].	0.0450		
T12	Query an employee's salary history in chronological order.	0.0600		
T13	Create an HTML template for managing CRUD operations on the "departments" table.	00444	TT4. Usage of Handlebars templates	MA2. Company Website
T14	Create an HTML template for managing CRUD operations on the table of employees who are department managers.	0.0889		
T15	Use "SelectManagersDepto (T4)" with its fields to create a template for cards of current managers in each department.	0.0889		
T16	Use "SelectEmployeesListDepto (T5)" with its fields to create a template for pills of current employees in a department.	0.0889		
T17	Create a diagram to display the current number of men and women in the company.	0.0444		
T18	Create a diagram to display the current number of employees in each department.	0.0444		
	Total	1		

3.2.2. Quantitative

This section provides a detailed analysis of the quantitative results obtained from the experimental session involving the 39 participants as well as from the usability questionnaire.

Table 5 displays the outcomes of all participants' execution of the 18 tasks within the experiment. The columns represent the various tasks, while the rows identify the participants. Additionally, both task and participant overall averages and standard deviations are showcased in the final rows and columns, respectively.

Table 5. Outcomes of all participants' execution of the 18 tasks in the experiment.

P#	Tasks																			
	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**T11**	**T12**	**T13**	**T14**	**T15**	**T16**	**T17**	**T18**	**Avg**	**SD**
P1	100	66.7	33.3	66.7	66.7	50	50	50	66.7	66.7	33.3	75	0	0	0	0	0	0	40.3	32.9
P2	100	66.7	33.3	33.3	33.3	50	50	50	33.3	0	33.3	25	0	0	40	10	0	0	31	27.1
P3	100	33.3	100	66.7	66.7	50	50	50	66.7	66.7	33.3	75	60	70	60	30	80	80	63.2	20.2
P4	100	33.3	33.3	33.3	0	25	25	50	33.3	33.3	0	25	80	70	70	80	40	40	42.9	27.3
P5	100	66.7	33.3	33.3	33.3	25	50	50	66.7	33.3	33.3	100	20	40	10	10	20	20	41.4	26.8
P6	100	100	100	100	100	75	75	100	100	100	100	75	100	90	90	100	100	100	94.7	9.6
P7	100	0	66.7	66.7	66.7	50	75	100	66.7	66.7	0	50	60	70	50	40	40	0	53.8	29.7
P8	100	100	100	100	100	75	75	100	100	100	100	75	100	100	100	80	100	100	94.7	10.2
P9	100	100	100	100	100	75	75	100	100	100	100	75	100	100	100	90	80	100	94.2	10.2
P10	100	0	100	66.7	66.7	50	50	50	66.7	66.7	33.3	50	60	60	60	10	80	80	58.3	25.8
P11	100	100	100	100	100	75	75	100	100	100	100	75	100	90	90	90	80	80	91.9	10.3
P12	100	66.7	33.3	33.3	33.3	50	25	50	33.3	33.3	33.3	25	60	70	40	40	60	0	43.7	22.1
P13	100	100	100	100	100	75	75	100	100	100	100	75	100	100	90	90	100	100	94.7	9.6
P14	100	100	66.7	66.7	66.7	50	50	100	100	66.7	66.7	75	20	0	0	0	0	0	51.6	38.6
P15	100	100	66.7	66.7	33.3	75	50	100	66.7	66.7	0	50	100	90	0	50	80	0	60.8	34.2
P16	100	66.7	33.3	66.7	33.3	50	50	50	33.3	33.3	33.3	50	60	70	30	30	80	60	51.7	19.8
P17	100	66.7	33.3	33.3	33.3	25	25	50	66.7	33.3	33.3	100	0	0	0	0	0	0	33.3	32.7
P18	100	100	66.7	100	66.7	50	75	100	66.7	66.7	66.7	50	100	50	50	0	40	100	69.4	27.5
P19	100	100	100	66.7	100	75	75	100	100	66.7	0	75	100	0	70	50	100	0	71.2	36.3
P20	100	100	100	100	100	75	75	100	100	100	100	75	100	80	80	80	60	60	88.1	14.8
P21	100	33.3	33.3	66.7	66.7	50	50	50	66.7	66.7	33.3	100	20	50	20	10	20	0	46.5	28.2
P22	60	33.3	33.3	33.3	33.3	25	25	50	33.3	33.3	33.3	50	80	50	0	60	40	40	39.6	17.3
P23	100	100	66.7	66.7	66.7	75	50	100	66.7	66.7	66.7	50	60	60	80	0	0	0	59.7	31.3
P24	100	66.7	66.7	66.7	66.7	25	50	50	66.7	66.7	33.3	25	60	70	0	20	60	20	50.7	25
P25	100	100	100	66.7	66.7	75	75	100	66.7	66.7	100	50	60	0	70	60	20	20	66.5	29.6
P26	100	100	100	100	100	75	75	100	100	100	100	75	100	90	90	90	80	80	91.9	10.3
P27	100	0	0	0	0	0	0	100	66.7	66.7	100	50	40	60	70	30	40	40	42.4	37
P28	100	66.7	33.3	66.7	66.7	25	50	50	66.7	0	0	75	60	70	30	10	60	40	48.3	27.4
P29	100	33.3	100	66.7	66.7	50	50	50	66.7	66.7	33.3	75	60	70	60	10	80	80	62.1	22.6
P30	100	66.7	66.7	66.7	66.7	50	50	50	100	66.7	33.3	75	100	90	40	30	100	100	69.5	24.3
P31	100	66.7	100	66.7	66.7	50	50	50	100	66.7	33.3	75	60	70	40	30	100	0	62.5	27.3
P32	100	100	66.7	100	66.7	50	75	100	66.7	66.7	66.7	50	80	50	50	0	40	40	64.9	26.1
P33	100	33.3	33.3	33.3	33.3	25	25	50	33.3	33.3	33.3	25	40	60	20	10	20	20	34.9	19.9
P34	100	33.3	66.7	66.7	33.3	25	50	50	66.7	66.7	33.3	75	20	50	40	10	20	0	44.8	25.7
P35	100	33.3	33.3	33.3	0	25	25	50	33.3	33.3	0	25	100	90	50	70	40	40	43.4	29.4
P36	100	100	100	100	100	75	75	100	100	100	0	75	100	100	80	80	40	40	81.4	28.3
P37	100	100	66.7	100	66.7	75	50	50	100	66.7	66.7	100	60	70	40	30	60	40	69	23
P38	100	66.7	33.3	33.3	33.3	50	50	50	66.7	66.7	33.3	75	0	0	0	0	0	0	36.6	31.5
P39	100	66.7	66.7	66.7	33.3	50	50	50	66.7	33.3	33.3	25	60	70	50	30	100	100	58.4	23.9
Avg	99	68.4	65.8	66.7	59.8	51.3	53.2	71.8	71.8	62.4	47.1	62.2	63.6	59.5	47.7	37.4	52.8	41.5	60.1	
SD	6.4	32.4	309	26.5	29.8	20.6	19.2	25.1	23.6	26.7	34.8	22.9	33.7	32.2	31.7	32.9	34.5	38.5		31.5

The values denote the percentage of task completion success based on their level of difficulty, as indicated in Table 5. Colors have been utilized to differentiate success levels, ranging from green for higher results to red for lower outcomes.

The results highlight that tasks T1, T8, and T9 achieved the highest average success percentages, scoring 99 (SD = 6.4), 71.8 (SD = 25.1), and 71.8 (SD = 23.6), respectively. Particularly noteworthy is the low standard deviation for task T1, where only participant P22 did not achieve 100% success in the task. These tasks were linked to the initial configuration of SQLRho (TT1) and query operations within the 'employees' table (TT3).

In contrast, tasks T16, T18, and T15 obtained the lowest average success percentages, scoring 37.4 (SD = 32.9), 41.5 (SD = 38.5), and 47.7 (SD = 37.7), respectively. Unlike tasks with high scores, these tasks exhibited high standard deviations, primarily due to over 10 participants obtaining percentage ratings between 0 and 20. All these tasks shared the usage of Handlebars templates (TT4) based on operations within the 'departments' (TT2) and 'employees' (TT3) tables.

Regarding individual participant results, it is evident that over 60% achieved high levels in the average task success percentage, i.e., results exceeding 50%. For instance, participants P6, P8, P9, P11, P13, and P26 scored above 90%, all with standard deviations below 11. Conversely, participants P2, P17, and P33 achieved the lowest average success percentages, scoring 31 (SD = 27.1), 33.3 (SD = 32.7), and 34.9 (SD = 19.9), respectively. These lower results are primarily attributed to low scores in tasks involving the use of Handlebars templates (TT4).

These outcomes not only offer insights into participants' strengths and weaknesses in executing specific tasks but also suggest opportunities for continuous improvement. Providing additional resources, specific training on Handlebars template usage, and alternative approaches to address identified challenges could benefit the future execution of similar tasks.

Table 6 presents the success rate and completion of tasks carried out by the participants in the experiment. The rows represent the 18 tasks that the participants performed during the study (see Table 4 for additional details). In the "Task Success Rate" columns, the results of successful task execution, their types, and associated main activities are displayed.

It is noteworthy that the development of the SQLRho program (MA1) achieved a higher average success rate than the corporate website development (MA2), recording 65 (SD = 28.6) and 50.4 (SD = 34.9), respectively. It is relevant to mention the high variability with standard deviations above 25 in both cases.

Among the task types, those related to configuring the SQLRho program with the 'employees' database (TT1) and operations in the 'employees' table (TT3) obtained the highest ratings, with 99 (SD = 6.4) and 63 (SD = 26.6), respectively. All these results surpassed the 50% threshold, considered positive, especially given the diversity and difficulty of the tasks.

Generally, tasks involving the use of Handlebars templates obtained lower success rates due to their higher level of difficulty. These tasks required the utilization of previous task outcomes as well as integrating templates, CRUD operations, and diagrams for website development.

It is observed that activities related to the initial setup of the SQLRho program (MA1) stand out with more consistent and higher performance compared with corporate website development (MA2). This discrepancy might suggest a greater familiarity or mastery by the participants in program configuration rather than web design and development.

Regarding the task completion rate, Table 6 identifies the number of incomplete tasks (considered as null scope in task execution), completed tasks (with some level of success-

ful execution), and the rate (calculated as the ratio between the number of incomplete and completed tasks). It is notable that tasks T1, T8, and T9 achieved a 100% completion ratio. Moreover, 94% of the tasks achieved a completion ratio higher than 80%, which is considered positive given the variety and difficulty of the evaluated tasks. Toward the conclusion of our experimental session, participants were asked to complete a comprehensive 30-question questionnaire to evaluate the usability of our proposal. This questionnaire was designed to assess various aspects such as utility, ease of use, ease of learning, and satisfaction.

Table 6. Success rate and task completion by experiment participants.

Tasks	Task Success Rate				Task Completion Ratio		
	%	SD	Task Type% (SD)	Main Activity% (SD)	N° of Uncompleted Tasks	N° of Completed Tasks	Completion Rate (%)
T1	99	6.4	TT1. 99 (6.4)	MA1. 65 (28.6)	0	39	100
T2	68.4	32.4	TT2. 60.9 (26.4)		3	36	92.3
T3	65.8	309			1	38	97.4
T4	66.7	26.5			1	38	97.4
T5	59.8	29.8			3	36	92.3
T6	51.3	20.6			1	38	97.4
T7	53.2	19.2			1	38	97.4
T8	71.8	25.1	TT3. 63 (26.6)		0	39	100
T9	71.8	23.6			0	39	100
T10	62.4	26.7			2	37	94.9
T11	47.1	34.8			7	32	82.1
T12	62.2	22.9			0	39	100
T13	63.6	33.7	TT4. 50.4 (34)	MA2. 50.4 (34.9)	4	35	89.7
T14	59.5	32.2			7	32	82.1
T15	47.7	31.7			7	32	82.1
T16	37.4	32.9			7	32	82.1
T17	52.8	34.5			6	33	84.1
T18	41.5	38.5			13	26	66.7
	Total			60.1 (31.5)	63	639	90.1

Figure 12 depicts the outcomes derived from analyzing the questionnaire responses, showcasing the average scores (ranging from 1 to 5) obtained for each dimension. Additionally, these results are presented alongside error bars, representing the standard deviation ($\pm\sigma$).

Given the proximity of the mean results across the four dimensions, we delved into a potential correlation among these values. Consequently, a calculation was conducted to gauge the correlation coefficient between the assessed dimensions. This analysis sought to quantify the extent of linear association among these dimensions. To facilitate this calculation, an initial step involved constructing a matrix to explore correlations across all dimensions, allowing the identification of those with stronger connections (demonstrating a correlation coefficient threshold exceeding 70%). Subsequently, scatter plots were generated for the mean scores of the selected dimensions, facilitating comparison and computation of the linear correlation coefficient (R^2). This coefficient ranges between 0 and 1, indicating a stronger linear relationship between dimensions as it approaches 1.

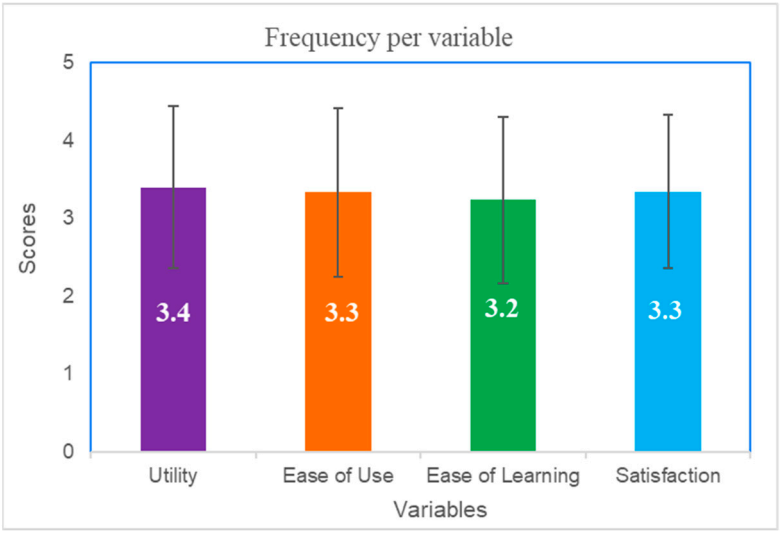


Figure 12. RestRho usability. Mean score for each dimension that is measured in values between 1 and 5, including error bars (±σ).

The most substantial correlation uncovered emerged between the ease of use and ease of learning dimensions, registering a robust linear correlation coefficient of $R^2 = 0.8669$, denoting a positive correlation between these aspects (see Figure 13). This correlation implies that users experience heightened ease of use while utilizing RestRho, particularly when the tool offers increased accessibility in mastering its functions.

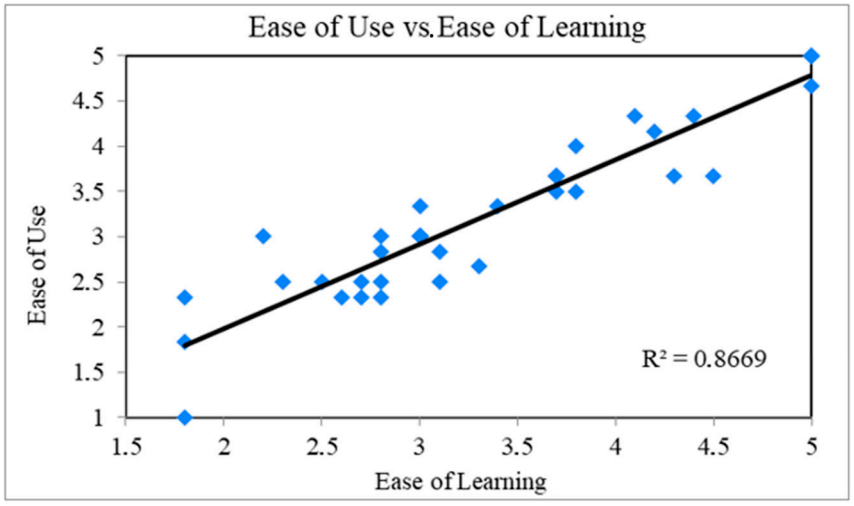


Figure 13. Correlation between the dimensions ease of use and ease of learning.

Figure 14 showcases the second most notable correlation observed, between utility and ease of use, unveiling a compelling linear correlation coefficient of $R^2 = 0.8118$. This statistical observation underscores a noteworthy trend: as RestRho exhibits enhanced user-friendly attributes, users tend to perceive greater practicality in the tool. This correlation underscores the pivotal role of usability in bolstering the perceived value of RestRho among users, accentuating the significance of an intuitive and seamless user experience in maximizing the tool's overall utility.

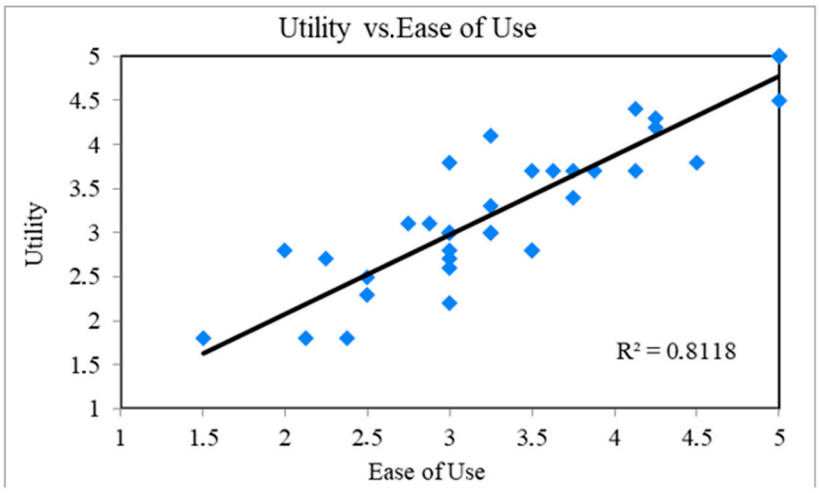


Figure 14. Correlation between the dimensions utility and ease of use.

3.2.3. Qualitative

The purpose of the proposal is to permit software engineers and domain experts to write the least code possible with a faster outcome in creating APIs and web pages without any complications. To determine if this was achieved, we conducted a qualitative study followed by an experiment that was introduced with final-year students pursuing a degree in computer engineering. The purpose of the study was to gather the opinions from the students of their thoughts on the proposal, specifically compared with other applications on the market. The fundamental evaluation sought from the students was what additional benefits the proposed RestRho provides to what already exists.

The qualitative study was conducted via a usability questionnaire that was based on [50]; 39 students responded to the 7 questions presented in the table below (see Table 7).

Table 7. Usability questionnaire deployed to 39 students following an experiment to gauge the validity of the proposed RestRho.

No.	Question
1	What do you think about the DBRestRho and SQLRho languages for creating web pages on the proposed RestRho? Do you think this could offer additional benefits compared to other RESTful API Server options?
2	What do you think would be the main advantages of being able to create web pages using the proposed RestRho?
3	What do you think would be the main advantages of being able to create web microservices using the proposed RestRho?
4	Imagine that you are developing a web project, and you need to use templates to speed up the process. How do you think Template Engines (e.g., Handlebars) could make this job easier for you?
5	Would you like to share any idea or additional suggestion about the functionalities proposed in the RestRho to create web pages?
6	What aspects do you consider negative of the RestRho?
7	What aspects do you consider positive of the RestRho?

From the usability questionnaire, a total of five themes emerged from the responses gathered from 39 students. The themes are “Validity”, “Utility”, “Usability”, “Ease”, and “Security”. Provided below is an analysis of the usability questionnaire responses, including an analysis and description of each of the themes.

Validity

Validity intends to determine how well the proposed RestRho addresses the needs of software engineers and domain experts. This theme specifically explores whether the users' opinions reflect the intended benefits of the proposal: does the proposed RestRho provide additional benefits to what already exists. In addition to benefits, the questionnaire explored the positive features of RestRho.

Question one of the usability questionnaires (see Table 7) resulted in both positive and negative responses regarding the RestRho validity (see Figure 15). Positive and negative responses determined the validity of the server displayed as valid or invalid, respectively, whereas blank responses or responses such as *"I have no opinion"* were deemed neutral, as they were not classified as positive nor negative.

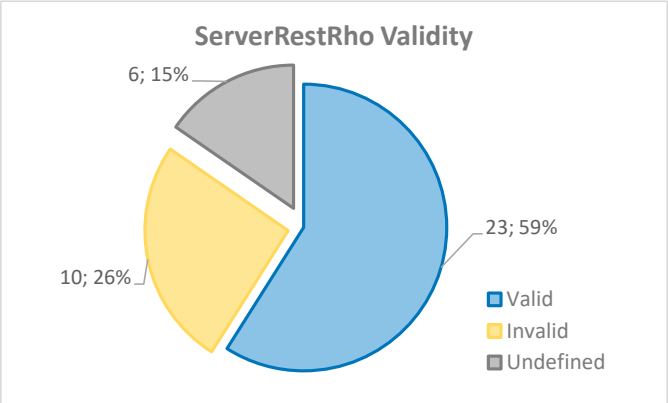


Figure 15. Validity of the proposed RestRho.

The majority of students indicated the validity of RestRho, with many indicating the benefits of utilizing DBRestRho and SQLRho languages as they are based on common languages such as SQL, JSON, and JavaScript.

> *"In my opinion, the DBRestRho and SQLRho languages are quite easy to understand and implement. Other tools require learning an entire language from scratch, however, being based on JavaScript and JSON, the language base one already has and only has slight differences."* (Student 4)

> *"By using DBRestRho and SQLRho on the RestRho, some additional benefits can be gained compared to other RESTful API server options"* (Student 5)

> *"I think they make efficient use of SQL and JSON for the creation of Pages. Yes, because there are other options that require a lot more writing to achieve the same thing."* (Student 35)

However, five of the ten students expressing a negative response to question one indicated that with additional training and documentation, their experience could be greatly improved.

> *"It is complicated to learn and implement, I think it has potential but more work is needed on the documentation and tools to learn how to use it."* (Student 21)

> *"In my opinion, from what I was able to do it is very interesting, but more practice is needed to be able to create web pages, and I think it would bring some additional benefits."* (Student 33)

> *"There is not much documentation on the web, it takes time to get familiar with the language"* (Student 37)

This indicates that the lack of learning resources heavily impacted the validity of RestRho and is an important improvement to consider in future works.

Utility

Utility aims to understand the usefulness and added benefit that the DBRestRho and SQLRho programming languages provide to software engineers and domain experts in using RestRho to create APIs and web pages. The main advantages of both programming languages were analyzed, including the value that RestRho brings to building web pages. Furthermore, the study identified the positive aspects of RestRho.

The main benefits indicated include speed, efficiency, flexibility, and modularity.

> *"Microservices allow you to split an application into smaller, independent components. Using RestRho to create web microservices could facilitate application modularity and scalability, as each microservice could have its own logic and responsibilities specific."* (Student 17)

Additionally, useful functionalities include reusable templates, microservices, and database integration.

> *"Efficiency, since it does not require too much code to make a request"* (Student 19)

> *"Flexible database integration"* is a positive aspect of RestRho. (Student 21)

> *"I would not have to create what I want to show several times, just calling that template will be enough to be able to visualize its function."* (Student 23)

However, some students questioned the scalability of RestRho for larger and more complex projects. There were also concerns with limited compatibility with existing tools and the associated consequences.

> *"If the RestRho is not compatible with certain existing tools, frameworks or systems, it may have difficulty integrating with other infrastructure components or taking advantage of popular technologies."* (Student 17)

> *"From what I could observe it may be the limited scalability that was still seen, some limited complexity in some complex web projects"* (Student 33)

Usability

Usability aims to identify if the students were able to build Rest APIs and webpages without any complications while using RestRho. It analyzed the functionality of the proposed RestRho and identified the benefits such as how it would support software engineers and domain experts to easily develop APIs and webpages. Additionally, the study identified functionalities of RestRho that need improvements.

Many responses indicated the use of the programming languages (requiring prior knowledge) were intuitive and easy to use, once they understood the workflow.

> *"The ease of use. Much of the homework was done with the knowledge that I simply had up to now."* (Student 1)

> *"I find it quite useful once the workflow is known."* (Student 30)

However, in order to achieve this, some students noted that there was an initial steep learning curve that could be resolved with additional documentation, examples, and tutorials.

> *"Little documentation."* (Student 6)

> *"I wish there was more documentation to look at in case there is a concept that I do not understand."* (Student 8)

> *"The documentation should be a bit more explanatory."* (Student 18)

"Lack of clear documentation and usage examples." (Student 21)

Additionally, inconsistencies in syntax and behavior were highlighted, impacting the usability of RestRho.

"Having to learn new syntax and use more than one program to see results." (Student 12)

Ease

The ease of use of the proposed RestRho is ascertained by this theme as it explores the ease of navigation and functionality. Moreover, negative aspects of RestRho are explored and analyzed.

The majority of respondents indicated the quick development of web pages in comparison with using other tools, as RestRho required less coding.

"The server is much faster and its syntax is easier to handle. There is obviously an advantage against Java Spring, for example." (Student 1)

"In general it is quite complete and fast." (Student 7)

"Speed and above all his intuitiveness to do things." (Student 13)

"The server is much faster and its syntax is easier to handle. There is obviously an advantage against Java Spring, for example." (Student 18)

"It can be done faster unlike other programs" (Student 27)

"Speed to create them [Web Microservices]." (Student 30)

"It is faster" (Student 39)

Additionally, the ease of creating CRUD operations and making database queries was highlighted as a key advantage.

"Simplicity and Fast code creation (crud, services, login)" (Student 6)

"It delivers a framework, where you only have to worry about defining what is necessary." (Student 30)

However, some respondents noted error handling challenges, which made troubleshooting difficult.

"Error handling is complicated due to JavaScript; this could be improved in the future." (Student 21)

"Mainly error handling, it would be useful to know what I'm missing or what I've done wrong, for example, if I misspelled the name of a table, it could indicate that "table "table1" does not exist in the database" or something for the style" (Student 32)

Security

Security addresses issues associated with security, performance, and authentication. While there was no specific question addressing security, it was a common theme that arose in questions two, five, and six (see Table 7), suggesting that security is both an advantage of RestRho yet also a negative requiring improvement.

When identifying the advantages of using RestRho to create web pages, flexibility and security were identified as the main advantages.

"Flexibility—Control over the page—Security and personalization" (Student 16)

Yet, security was also raised as a negative aspect of RestRho, as it may create dependencies on other libraries and tools, creating security issues.

"If the RestRho uses third-party libraries or tools, there is a risk of being dependent on their maintenance and updating. If these dependencies are not regularly updated or become outdated, there could be security issues or lack of long-term support." (Student 5)

In light of these negative aspects, the qualitative study also provided opportunities for improvement in the areas of authentication and security.

"Consider including authentication and authorization functionalities in the RestRho." (Student 5)

"Focus on security: Make sure that the RestRho implements strong security measures to protect web pages and associated data. This can include authentication, authorization, data encryption and prevention of attacks such as SQL injections and XSS (cross -site scripting)." (Student 17)

Overall, students identified the benefits of RestRho as it supported them to quickly create web pages. However, there were concerns raised about the ease of using the platform (the learnability and its learning curve) and its security. More learning resources, such as documentation and tutorials, could help address these concerns.

4. Discussion of the Findings

This section integrates the findings from both the code validation and the empirical validation to provide a comprehensive analysis of RestRho's performance and usability.

4.1. Code Validation

The code validation of RestRho through various software metrics reveals significant insights into its quality and maintainability. Specifically, the cyclomatic complexity and Halstead metrics indicate a system with low complexity and high maintainability. The average per-function cyclomatic complexity (CNN) for RestRho is well below the threshold, suggesting that the system is easy to understand and maintain.

Halstead metrics further demonstrate RestRho's superiority, showing the lowest difficulty (HD), effort (HE), and error estimate per module (HB) among the components analyzed. This comparison with other components and frameworks, such as those in RhoArchitecture (CRA) and the frameworks used by RestRho (FRR), underscores RestRho's superior productivity and ease of maintenance.

Additionally, the documentation quality of RestRho is notable, with comment lines (CLOC) averaging 34%, surpassing the 25% threshold, which ensures that the codebase is well documented and easier for developers to navigate. This is particularly evident in the ServerRestRho component, which has the highest documentation rate at 43%. The maintainability index (MI) for all components and frameworks analyzed shows a "good maintainability" score (>85), with RestRho's components averaging higher than both CRA and FRR, demonstrating its robust and reliable nature. Overall, these findings confirm that RestRho is a well-structured, maintainable system, positioning it favorably compared with other components and frameworks analyzed.

4.2. Empirical Validation

Both quantitative and qualitative studies were carried out to explore the proposed RestRho, determining if it enabled software engineers and domain experts to build web pages faster with the least code possible. The quantitative study included both an experiment and a questionnaire, followed up by a qualitative usability questionnaire.

The experiment illustrated that as the complexity became more difficult, it directly impacted the quality of the outcome and the ability to complete each task, which was expected. This is evident by analyzing the sub-categories (see Table 4) demonstrating a high

completion rate at the beginning of the set of tasks, which steadily decreased as tasks became more complex.

This was compounded by both the prior understanding of the DBRestRho and SQL-Rho languages versus the need for additional learning in specific functionality in RestRho. Students were able to easily apply their knowledge to producing higher-quality outcomes when prior knowledge existed. However, as the tasks became more complex and more native to the platform, the students were required to learn new functionalities via tutorials and written documentation.

Nevertheless, in the qualitative study, many students raised their concerns about the steep learning curve. A total of 12.8% of respondents implied a low validity of RestRho because of a lack of documentation and the need for additional tutorials. This indicates that with additional time (training and practice) and further detailed documentation, the validity score will increase. As a result, the limited learning resources impacted the learning curve and, in turn, both the success rate and completion rates of latter tasks within each sub-category.

Additionally, the qualitative study revealed that the absence of error handling proved to be challenging when it came to troubleshooting issues and finding resolutions. Additionally, the usability questionnaire, conducted during the quantitative study, demonstrated a dispersed response range, indicating that the ease of use of RestRho must be improved.

Yet, in the same usability questionnaire, which explored utility, ease of use, ease of learning, and satisfaction, there is a direct correlation between ease of learning and ease of use. This is an expected result; when a platform is easier to learn, it becomes easier to use. Likewise, when a platform is easier to use, it becomes easier to learn. A direct correlation between the two is expected. The qualitative study supports this where the majority of respondents indicated the quick development in comparison with alternative tools as a direct result of ease of use. It is, then, evident that RestRho permits software engineers and domain experts to quickly create web pages while writing the least code possible, the purpose of this proposal.

While the ease of use and usability indicate that there is a steep learning curve, students expressed in the qualitative feedback that once the workflow was known, it was easy to implement, especially because they could use existing languages. As previously identified, this steep learning curve could be solved with more documentation, examples, and tutorials.

The same correlation exists between ease of use and utility. The more useful the application was to the student, the easier it became to use (implying that when the motivation to learn is high, ease of use subsequently increases). The purpose of this proposal was to enable software engineers and domain experts to quickly create web pages while writing as little code as possible. As main drivers in selecting a platform, it is imperative that these attributes were achieved. The qualitative study demonstrated that the main benefits were speed, efficiency, flexibility, and modularity. As a result, by focusing on the motivations that increase ease of use (the usefulness of the application) alongside the main drivers in selecting a platform (speed and efficiency), it is evident that RestRho successfully achieves what has been proposed, faster delivery with less coding.

This is further evident in the usability questionnaire of the qualitative study, where 59% of students indicated the validity of RestRho, with a further 12.8% of students indicating that validity would increase with additional documentation and training. A further 15% of students had a neutral result for validity, such as “I have no opinion”.

Overall, both the quantitative and qualitative studies have demonstrated that RestRho supported students to deliver results faster with less time writing code. Concerns around the ease of learning have been made evident, which can be resolved with more documen-

tation and tutorial sessions. Nevertheless, the validity and utility themes of the qualitative analysis verified that RestRho successfully achieves its proposed objectives.

4.3. Integration of Findings

The findings from both the code validation and the empirical validation give a comprehensive view of RestRho's strengths and areas for improvement. The code validation showed that RestRho is well structured, maintainable, low-complexity, and well documented compared with other frameworks. The empirical validation confirmed these aspects and highlighted the system's effectiveness and ease of use for rapid web development. However, the empirical study also revealed challenges related to the learning curve and the need for better documentation and tutorials. These issues need to be addressed to enhance the user experience and accessibility of RestRho.

The code validation and empirical validation show that RestRho is a powerful tool for minimal coding web development. The high code quality and positive user feedback indicate that RestRho meets its design objectives. Improving documentation and providing more learning resources can further increase its adoption and success.

4.4. Comparison with State-of-the-Art DSL Tools

While Table 1 provides a functional overview of the most widely used DSLs for RESTful API specification—such as RAML, Swagger/OpenAPI, WADL, and API Blueprint—it lacks a benchmark-oriented discussion grounded in practical development scenarios. To address this, we propose a comparative evaluation based on three real-world use cases frequently encountered in web API engineering: (i) rapid deployment of CRUD operations, (ii) dynamic result transformation through integrated template engines, and (iii) extensibility and maintainability over time. In these scenarios, RestRho, through its JSON-based DSLs (DBRestRho and SQLRho), provides higher abstraction, seamless modularization, and native support for Handlebars templates, enabling faster and more customizable API generation. In contrast, existing tools often rely on rigid specifications and require additional frameworks to achieve equivalent flexibility, increasing both complexity and coupling.

From a software engineering standpoint, the proposed solution outperforms the state of the art in both structural and empirical dimensions. RestRho exhibits low cyclomatic complexity (CNN = 1.4) and a high maintainability index (MI = 121.1), reflecting simplified logic and strong sustainability over time. Furthermore, empirical data collected from 39 final-year computer engineering students shows a 100% task completion rate in foundational API setup activities (T1, T8, T9), validating the tool's effectiveness and ease of use. These findings suggest that RestRho not only is viable as a domain-specific solution but also is competitive against mature alternatives when developer productivity, extensibility, and runtime customization are critical requirements.

5. Threats to Validity

In this section, we discuss the threats with respect to the conclusion's validity, internal validity, external validity, and construct validity.

5.1. Integration of Findings

For the validation of the RestRho implementation, we used software metrics that provide an objective assessment of various features of an information system. Thus, no assumption was violated. For the experimental analysis, at the quantitative level we used non-parametric statistical tests, i.e., no assumptions were made about the underlying distribution of the obtained samples, so there would be no violation of any assumptions either.

However, this may imply a low statistical power, which may lead to a threat to the validity of the conclusion, although the conclusions are also supported by the qualitative analysis.

Regarding the reliability of the measures of the quantitative part of the experiment, there might be some threat, as the measures of success rate and task completion by the participants are based on the score given by the authors of this study. To mitigate this threat, a detailed rubric was defined in advance for the whole set of tasks; although this approach does not completely eradicate the threat to the reliability of the scores, it minimizes the risk of bias.

There are no threats due to the heterogeneity of the subjects, as the study group has a similar general level of skills and knowledge that would have no effect on the observed outcome. On the other hand, random irrelevancies in the experimental setting may be a threat to the validity of the findings of this study; because the participants could perform the tasks outside of the classroom, there was no way to prevent or track any event that could have had an effect on the results, for example, cheating on a result or having a different partner perform the activity. While no explicit measures were taken to ensure that all participants performed the study in a controlled environment, there are still features in our study design that could serve as a mitigation of this threat, such as being able to compare results to check that they were not the same and the assessment of the knowledge gained.

5.2. Threats to the Internal Validity

With relation to the threats that could result from learning effects and burnout, they may be concerned with the learning curve and the time for task execution. In the first case, for this research, learning effects were not an issue because of the homogeneity of the group, so no measure was taken. Execution time was not used as a metric; instead, the level of completion of each task was assessed.

Motivation was considered another potential challenge. Each participant presumably reacts differently to the experiment, and some participants may perform poorly, especially if they alternate between the experiment and other activities. This challenge cannot be entirely mitigated. However, with the assistance of the final usability questionnaire and the qualitative assessment questions, the analysis was complemented, and the impact of this challenge was reduced.

5.3. Threats to the External Validity

An external threat was identified concerning the generalizability of the experimental results to broader user populations. This potential threat arises when the selected participant sample does not adequately represent the target audience. Ideally, the study would involve a direct sample of the intended user population. Therefore, our sample population has some limitations for generalization to the target users, who would preferably be domain experts or professional developers. However, since the students were in their final course of a computer science degree, and the JSON-DSL RestRho was created for the implementation of RESTful APIs, which is a kind of implementation known by this audience, we can gather reasonably reliable empirical evidence on the implementation and usage of the solution.

Nevertheless, caution is required when extrapolating these findings to professional environments. Future replications of the study will include participants from industrial settings to enhance external validity and broaden the evaluation of the tool's practical applicability.

5.4. Threats to Construct Validity

This threat relates generalizing the findings to the concept and/or theory underlying the experiment. Monomethod bias was avoided, as the validation was performed using several ways of assessing the results, with a practical activity, a questionnaire for quan-

titative analysis, and a questionnaire for qualitative analysis. Another potential threat is hypothesis guessing, since participants guessing about the purpose of the study may adapt their behavior. This threat was addressed by communicating the purpose of the study as openly and clearly as possible and explicitly asking participants to use the tool with a critical and learning attitude.

Finally, using questionnaires may bias the results on the satisfaction response variable. However, this approach is used in many studies and has become a reliable mechanism for this measure.

6. Conclusions and Future Work

DSLs enable the construction of fast, robust, and flexible software applications with high productivity for both software engineers and domain experts due to their high level of abstraction. In particular, JSON-DSLs allow us to simplify the definition of DSLs using the JSON standard. Our RhoArchitecture includes an evaluation engine, a programming model, and an integrated IDE Web, which facilitates the creation of JSON-DSLs.

This article has introduced the RestRho case study, a JavaScript component designed to dynamically define and implement HTTP requests on a RESTful NodeJS server. RestRho specifies and implements two JSON-DSLs using RhoArchitecture (DBRestRho and SQLRho) to define and execute requests on a ServerRestRho. To validate RestRho, users created a specific RESTful API for employee management using these two languages.

To validate the implementation of RestRho, a set of software metrics was used to objectively measure several software features. The analysis demonstrates that RestRho reduces the complexity of the code, shows a low level of difficulty of the code together with high documentation quality, and it has low error estimates and exhibits good maintainability.

To validate the use of JSON-DSLs, an experiment was carried out with final-year students pursuing a degree in computer engineering to determine the quality of the generated code and the usability of the proposal. After conducting the experiment, both quantitative and qualitative studies were undertaken to further analyze effectiveness, efficiency, satisfaction, validity, utility, ease of use, and ease of learning. The quantitative evaluation of RestRho revealed high effectiveness for simpler tasks but a decline as task complexity increased. Efficiency, inferred from task completion rates, also dropped with higher task complexity. Satisfaction scores from the usability questionnaire were generally positive. The qualitative feedback supported these findings, highlighting quick development times and ease of use after the initial learning phase, validating RestRho's aim of enabling rapid web page creation with minimal coding.

Despite the learning challenges, students highlighted benefits such as speed, efficiency, flexibility, and modularity, with 58% affirming the validity of RestRho. The qualitative study also suggested that RestRho facilitates rapid development compared with alternative tools, aligning with the proposal's objective of enabling quick web page development with minimal coding.

The comparative evaluation shows that RestRho is a solid and efficient alternative to traditional DSLs for RESTful APIs. Its JSON-based approach, combined with native template engine support and a modular architecture, allows it to overcome the limitations found in tools like OpenAPI, RAML, and WADL. Both quality metrics and empirical results confirm that RestRho simplifies development, improves maintainability, and adapts well to real-world scenarios. This makes it a suitable choice for projects requiring speed, customization, and high-quality API development.

In future work, it would be interesting to have features to create a DVSL for the creation of these programs and for extending the grammar of the DBRestRho and SQLRho languages with the following features: (i) connections to more databases; (ii) expanding

the list of available template engines; (iii) applying concatenation of existing operations; and (iv) creating plugins for managing graphics and diagrams. Additionally, exploring RestRho for creating microservices based on REST architecture, enhancing flexibility and scalability, is promising. This involves implementing template engines and transformations for CRUD operations, aiming to increase efficiency and flexibility in web application development. To support broader adoption, future work will also include improving onboarding through clearer documentation, tutorials, and example projects to ease the initial learning curve observed in the empirical study. Finally, we plan to conduct new validations with more diverse and representative participants.

Author Contributions: Conceptualization, E.C., L.R., F.D.R., K.S., and F.J.; data curation, E.C., L.R., F.D.R., K.S., and F.J.; formal analysis, E.C., L.R., F.D.R., K.S., and F.J.; investigation, E.C., L.R., F.D.R., K.S., and F.J.; methodology, E.C., L.R., F.D.R., K.S., and F.J.; resources, E.C., L.R., F.D.R., K.S., and F.J.; software, E.C.; validation, E.C., L.R., F.D.R., K.S., and F.J.; writing—original draft, E.C., L.R., F.D.R., K.S., and F.J.; writing—review and editing, E.C., L.R., F.D.R., K.S., and F.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data available on request from the authors.

Acknowledgments: This work has been partially supported by the research R&P and development department from the UGround Company (http://www.uground.com/ (accessed on 10 July 2025).

Conflicts of Interest: Author Enrique Chavarriaga was employed by the company UGround Global S.L.; Author Kat Sorbello is founder of the company The Stella Way, 4118 Brisbane, Australia. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Fowler, M.; White, T. *Domain-Specific Languages*; Addison-Wesley Professional: Denver, CO, USA, 2010; ISBN 0-13-210754-6.
2. Mernik, M.; Heering, J.; Sloane, A.M. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv. (CSUR)* **2005**, *37*, 316–344. [CrossRef]
3. Ghosh, D. *DSLs in Action*; Manning Publications: London, UK, 2010; ISBN 1-935182-45-5.
4. Wu, H.; Gray, J.; Mernik, M. Grammar-Driven Generation of Domain-Specific Language Debuggers. *Softw. Pract. Exp.* **2008**, *38*, 1073–1103. [CrossRef]
5. Kelker, R.D. *Clojure for Domain-Specific Languages*; Packt Publishing: Birmingham, UK, 2013.
6. Hudak, P. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* **1996**, *28*, 196. [CrossRef]
7. Voelter, M. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*; Springer: Berlin/Heidelberg, Germany, 2013.
8. Pierantonio, A. *Domain-Specific Languages in Practice: With JetBrains MPS*; Springer: Cham, Switzerland, 2019; ISBN 978-3-030-73758-0.
9. Fawcett, J.; Quin, L.R.E.; Ayers, D. *Beginning XML*, 5th ed.; Wrox Press: Hoboken, NJ, USA, 2012.
10. ECMA ECMA-404: The JSON Data Interchange Syntax. Available online: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/ (accessed on 2 September 2021).
11. W3C Recommendation JSON-LD 1.1: A JSON-Based Serialization for Linked Data (W3C Recommendation 16 July 2020). Available online: https://www.w3.org/TR/json-ld/ (accessed on 12 December 2023).
12. Chavarriaga, E.; Jurado, F.; Rodríguez, F.D. An Approach to Build JSON-Based Domain-Specific Languages Solutions for Web Applications. *J. Comput. Lang.* **2023**, *75*, 101203. [CrossRef]
13. Hermida, J.M.; Meliá, S.; Arias, A. Xanui: A Textual Platform-Independent Model for Rich User Interfaces. *J. Web Eng.* **2016**, *15*, 45–83.
14. Barisic, A.; Amaral, V.; Goulao, M.; Barroca, B. Evaluating the Usability of Domain-Specific Languages. In *Software Design and Development*; IGI Global: Hershey, PA, USA, 2014; pp. 2120–2141.
15. ISO.org. Ergonomics of Human-System Interaction—Part 210: Human-Centred Design for Interactive Systems. In *Online Browsing Platform (OBP)*; ISO: Geneva, Switzerland, 2023.

16. Iung, A.; Carbonell, J.; Marchezan, L.; Rodrigues, E.; Bernardino, M.; Basso, F.P.; Medeiros, B. Systematic Mapping Study on Domain-Specific Language Development Tools. *Empir. Softw. Eng.* **2020**, *25*, 4205–4249. [CrossRef]
17. do Nascimento, L.M.; Viana, D.L.; Neto, P.A.S.; Martins, D.A.; Garcia, V.C.; Meira, S.R. A Systematic Mapping Study on Domain-Specific Languages. In Proceedings of the Seventh International Conference on Software Engineering Advances (ICSEA 2012), Lisbon, Portugal, 18–23 November 2012; pp. 179–187.
18. Hoffmann, B.; Urquhart, N.; Chalmers, K.; Guckert, M. An Empirical Evaluation of a Novel Domain-Specific Language—Modelling Vehicle Routing Problems with Athos. *Empir. Softw. Eng.* **2022**, *27*, 180. [CrossRef] [PubMed]
19. Bojinov, V. *RESTful Web API Design with Node.Js 10, Third Edition: Learn to Create Robust RESTful Web Services with Node.Js, MongoDB, and Express.Js*, 3rd ed.; Packt Publishing: Birmingham, UK, 2018; ISBN 9781788629140.
20. Wexler, J. *Get Programming with Node.Js*; Simon and Schuster: New York, NY, USA, 2019.
21. Lee, M.C. Software quality factors and software quality metrics to enhance software quality assurance. *Br. J. Appl. Sci. Technol.* **2014**, *7*, 3069–3095. [CrossRef]
22. Cordasco, G.; D'Auria, M.; Negro, A.; Scarano, V.; Spagnuolo, C. Toward a Domain-Specific Language for Scientific Workflow-Based Applications on Multicloud System. *Concurr. Comput. Pract. Exp.* **2021**, *33*, e5802. [CrossRef]
23. Johanson, A.N.; Hasselbring, W. Effectiveness and Efficiency of a Domain-Specific Language for High-Performance Marine Ecosystem Simulation: A Controlled Experiment. *Empir. Softw. Eng.* **2017**, *22*, 2206–2236. [CrossRef]
24. Frantz, R.Z.; Rehbein, M.H.; Berlezi, R.; Roos-Frantz, F. Ranking Open Source Application Integration Frameworks Based on Maintainability Metrics: A Review of Five-Year Evolution. *Softw. Pract. Exp.* **2019**, *49*, 1531–1549. [CrossRef]
25. Bocanegra, J.; Frantz, R.Z.; Roos-Frantz, F.; Basso, F.P. Evaluating the Semantic Transparency of Guaraná: A Domain-Specific Language for Enterprise Application Integration. *Softw. Pract. Exp.* **2022**, *52*, 967–983. [CrossRef]
26. Poltronieri Rodrigues, I.; de Borba Campos, M.; Zorzo, A.F. Usability Evaluation of Domain-Specific Languages: A Systematic Literature Review. In *Proceedings of the Human-Computer Interaction, User Interface Design, Development and Multimodality*; Kurosu, M., Ed.; Springer International Publishing: Cham, Germany, 2017; pp. 522–534.
27. Subramanian, H.; Raj, P. *Hands-On RESTful API Design Patterns and Best Practices: Design, Develop, and Deploy Highly Adaptable, Scalable, and Secure RESTful Web APIs*; Packt Publishing Ltd.: Birmingham, UK, 2019.
28. Malakhov, K.; Kurgaev, O.; Velychko, V. Modern RESTful API DLs and Frameworks for RESTful Web Services API Schema Modeling, Documenting, Visualizing. *Probl. Program.* **2018**, *4*, 59–68. [CrossRef]
29. Ferguson, R. JavaScript and Development Tools. In *Beginning JavaScript: The Ultimate Guide to Modern JavaScript Development*; Apress: Berkeley, CA, USA, 2019; pp. 11–24, ISBN 978-1-4842-4395-4.
30. API BluePrint. Available online: https://apiblueprint.org/ (accessed on 10 July 2025).
31. Flanagan, D. *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language*, 7th ed.; O'Reilly Media: Sebastopol, CA, USA, 2020.
32. Czech, G.; Moser, M.; Pichler, J. Best Practices for Domain-Specific Modeling. A Systematic Mapping Study. In Proceedings of the 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Prague, Czech Republic, 29–31 August 2018; pp. 137–145.
33. Yue, C.; Wang, H. A Measurement Study of Insecure Javascript Practices on the Web. *ACM Trans. Web (TWEB)* **2013**, *7*, 1–39. [CrossRef]
34. Rani, F.; Diez, P.; Chavarriaga, E.; Guerra, E.; de Lara, J. Automated Migration of Eugenia Graphical Editors to the Web. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, Virtual, 16–23 October 2020; pp. 1–7.
35. Chavarriaga, E. Modelo Programable para la Serialización y Evaluación de Modelos Heterogéneos en Clientes Web. Ph.D. Thesis, Repository Autonomous University of Madrid, Madrid, Spain, 2017.
36. Tahir, A.; MacDonell, S.G. A Systematic Mapping Study on Dynamic Metrics and Software Quality. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012.
37. Riaz, M.; Mendes, E.; Tempero, E. A Systematic Review of Software Maintainability Prediction and Metrics. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, FL, USA, 15–16 October 2009; pp. 367–377.
38. Jatain, A.; Mehta, Y. Metrics and Models for Software Reliability: A Systematic Review. In Proceedings of the 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), Ghaziabad, India, 7–8 February 2014; pp. 210–214.
39. Ahsan, S.; Hayat, F.; Afzal, M.; Ahmad, T.; Asif, K.H.; Asif, H.M.S.; Saleem, Y. Object Oriented Metrics for Prototype Based Languages. *Life Sci. J.* **2012**, *9*, 63–66.
40. Nguyen, V.; Deeds-Rubin, S.; Tan, T.; Boehm, B. *A SLOC Counting Standard*; University of Southern California, Center for Systems and Software Engineering: Los Angeles, CA, USA, 2007.
41. McCabe, T. A Complexity Measure. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 308–320. [CrossRef]

42. Halstead, M. *Elements of Software Science*; Elsevier: Amsterdam, The Netherlands, 1977.
43. Oman, P.W.; Hagemeister, J.; Ash, D. *A Definition and Taxonomy for Software Maintainability*; University of Idaho, Software Engineering Test Laboratory: Post Falls, ID, USA, 1991.
44. Chavarriaga, E.; Jurado, F.; Díez, F. An Approach to Build XML-Based Domain Specific Languages Solutions for Client-Side Web Applications. *Comput. Lang. Syst. Struct.* **2017**, *49*, 133–151. [CrossRef]
45. Gill, G.K.; Kemerer, C.F. Cyclomatic Complexity Density and Software Maintenance Productivity. *IEEE Trans. Softw. Eng.* **1991**, *17*, 1284. [CrossRef]
46. Lund, A.M. Measuring Usability with the Use Questionnaire12. *Usability Interface* **2001**, *8*, 3–6.
47. Davis, F.D. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Q.* **1989**, *13*, 319–340. [CrossRef]
48. Lin, H.X.; Choong, Y.-Y.; Salvendy, G. A Proposed Index of Usability: A Method for Comparing the Relative Usability of Different Software Systems. *Behav. Inf. Technol.* **1997**, *16*, 267–277. [CrossRef]
49. MySQL. Employees Sample Database. MySQL Documentation, Oracle Corporation. Available online: https://dev.mysql.com/doc/employee/en/ (accessed on 10 July 2025).
50. Rojas, P.L.A.; Truyol, M.E.; Calderon Maureira, J.F.; Orellana Quiñones, M.; Puente, A. Qualitative Evaluation of the Usability of a Web-Based Survey Tool to Assess Reading Comprehension and Metacognitive Strategies of University Students. In Proceedings of the Social Computing and Social Media. Design, Ethics, User Behavior, and Social Network Analysis: 12th International Conference, SCSM 2020, Held as Part of the 22nd HCI International Conference, HCII 2020, Proceedings, Copenhagen, Denmark,, 19–24 July 2020; pp. 110–129.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.