

Towards the Automatic Quality Evaluation of RESTful APIs Using Design Rule Violations

Daniel Abajirov

Student ID: 3309203

Timothy Ernst

Student ID: 3309012

Manuel Merkel

Student ID: 3310645

3rd Semester Softwareengineering M.Sc. 1st Semester Softwareengineering M.Sc. 2nd Semester Softwareengineering M.Sc.
Institute of Software Engineering (ISTE) Institute of Software Engineering (ISTE) Institute of Software Engineering (ISTE)
Email: st155165@stud.uni-stuttgart.de Email: st157344@stud.uni-stuttgart.com Email: st155131@stud.uni-stuttgart.de

Abstract—Context: Due to the importance of REST APIs, many tools were implemented to help developers to write better REST APIs and to avoid errors. Several authors and companies have defined design rules to which API definitions should adhere; among them is Massé with the book "The REST API Design Rulebook". This book has gained a lot of popularity among developers, and recently an empirical study provided evidence for the effectiveness of certain design rules created by Massé.

Objective: To help developers who follow Massé's design concepts, we implemented a tool to analyze OpenAPI definitions and provide feedback on design rule violations.

Methods: To evaluate the tool, we decided to run a benchmark-based analysis regarding robustness, effectiveness, and performance. To provide a better overview about effectiveness, we decided to investigate the metrics precision and recall.

Results: Starting with the robustness, we have an overall 98% success rate of analyzing OpenAPI definitions. On the effectiveness aspect, we manually examined a total of 1,596 rule violations for false-positives. This resulted in 140 false-positives, and therefore, in a precision of 91%. For recall, on the other hand, we analyzed a gold standard defined by experts. A total of 111 rule violations yielding 36 false-negatives, generating a recall of 68%. Lastly, for performance, we found that the tool performed particularly well for the majority of the files analyzed. We also found out that the CPU is not a bottleneck in performing the analysis with this tool, and that RAM usage plays a more important role.

Conclusion: The collected data suggests that the tool is quite robust to errors. For the effectiveness of the tool, a slightly larger margin lies between precision and recall. Still, the results indicate a positive effectiveness of the tool on the analyzed OpenAPI definitions. Regarding the performance aspect, the data from the results suggest that our tool would perform quite well for most OpenAPI definitions on most of today's computers. With these results, future research could investigate the quality of OpenAPI documents and how they behave with the number of paths in a definition.

I. INTRODUCTION

The importance of APIs is increasing. New technologies provide new opportunities for businesses and industries to be developed. Companies such as Google, Amazon, and Apple have transformed existing businesses and created new industries. An API is a crucial part of the cloud services landscape, connecting people and devices to the underlying platforms that

power each business and that tie these companies together behind the scenes. Due to the importance that APIs have, many tools were implemented to help the developers to avoid errors and write better APIs [1]. Specifications have been created to facilitate the creation of better APIs. This allows people and computers to discover and understand the subject's capabilities without access to source code, or documentation. The most widely used and recognized specification for web APIs is the OpenAPI Specification¹ (OAS).

REST(ful) APIs and web services [2] represent one of the commonly used ways to expose functionality via a well-defined and technology-agnostic interface. Despite their popularity, there is no standardized set of design rules for REST APIs. Different publications [3], [4] propose rules and best practices to build cleaner and more maintainable REST APIs. Massé, for example, wrote a book on RESTful API design rules [3], in which he gives definitions that span over multiple categories, such as the design of path segments in the URI, resources, HTTP request methods, responses, metadata design, and representation design. Although, existing work studied the detection of (anti-)patterns in REST APIs as well as the adherence to best practices (e.g., [5], [6]), a recent study analyzed the perception of industry experts regarding the importance and software quality impact of 82 REST API design rules [7]. Experimental evidence for the effectiveness of some of these rules has been provided by Pfaff [8], who studied if the understandability quality attribute is indeed linked with implementing these rules. Using these results, it is therefore possible to create next-generation tool-support for REST APIs that takes important or effective rules into account, and links them to quality attributes. However, some linters² that check for rule violations in an OpenAPI definition already exist. Nonetheless, none of these are based on Massé's rule set and the research based on it. Therefore, the goal of this study is to first analyze and select existing RESTful API design rules, and then, to create with this knowledge a tool-supported

¹OpenAPI Specification, <https://spec.openapis.org/oas/latest.html>

²OpenAPI Linters, <https://nordicapis.com/8-openapi-linters/>

approach with a Command Line Interface (CLI). This tool automatically detects and reports design rule violations in an actionable way, and links them to software quality attributes. Afterwards, the tool will be tested and evaluated regarding robustness, effectiveness, and performance.

To give a statement about these properties, we would like to answer three research questions concerning the implemented tool:

- RQ₁** How robust is the tool regarding the successful analysis of real-world APIs?
- RQ₂** How effective is the tool for correctly identifying rule violations?
- RQ₃** How fast and resource-efficient is the tool while analyzing real-world APIs?

The motivation behind the first research question is that we want to show how robust the implementation of the tool is without producing an error. The second research question focuses on evaluating the analysis of the tool and how effectively it works. Finally, with the third research question, we want to see how resource-consuming our tool is while performing API analysis.

The study will be conducted primarily in an exploratory and iterative fashion, switching between analysis, implementation, and testing. For the development of the tool, some form of prototyping will be used [9]. To be able to make a statement about the research questions, a large number of OpenAPI definitions from "apis.guru" has to be analyzed by the implemented tool. This results in a large data set, which will help us to evaluate the tool. In the analysis, we counted how many of the OpenAPI definitions generated an error in the tool to get an idea of the robustness of the tool. To determine the effectiveness of the analysis, the metrics recall and precision were calculated. Finally, to gather data about the performance of the tool, the time and the utilization of RAM and CPU were measured.

The paper is divided into eight chapters: first, chapter II explains the required technical background knowledge. Chapter III provides the state of the research and describes the work that inspired this study. The implementation of the CLI and how the design rules were implemented are discussed in chapter IV. Subsequently, chapter V describes the methods used in the study. The starting point of chapter VI is the analysis and evaluation of the gathered data. Chapter VII discusses the results of the study and addresses the threats to validity together with some future implementation suggestions, and lastly, chapter VIII concludes with a summary.

II. BACKGROUND

To fully understand the content of this paper and follow its logical thread, we must first explain five basic concepts, namely REST, API, REST API, Swagger, API design rules, and Natural Language Processing (NLP).

A. REST

The information from this section comes from the paper "Architectural Styles and the Design of Network-based Soft-

ware Architectures" by Fielding [10] and from the book "APIs: A Strategy Guide" by Jacobson, Brail, and Woods [1].

REST stands for representational state transfer and was created by computer scientist Roy Fielding. In his Ph.D. dissertation, Roy Fielding describes the principles behind the REST architectural style and the interaction constraints chosen to maintain those principles. There are six different types of constraints.

The first type is the *Client-Server*, which implements the principle of separation of concerns. This separation allows us to achieve portability of the user interface and improve the scalability of the server components. It also allows the components to evolve independently.

The second constraint, *Stateless*, enables stateless communication with the properties of visibility, reliability, and scalability. All these properties help to improve stateless communication between client and server.

The third constraint, *Cache*, is used to improve the network efficiency of the overall system. By adding the cache constraint, some interactions between the client and the server can be partially or completely removed, improving efficiency, scalability, and user-perceived performance by reducing the latency of the interactions.

Simplifying the overall system architecture is an important aspect of the fourth constraint, the *Uniform Interface*. The REST architectural style differs from other network-based styles by emphasizing a uniform interface between components. The software engineering principle of generality can be applied to interfaces and architecture, simplifying overall design and promoting decoupled service implementations. Several architectural constraints are required to achieve a uniform interface. In order to have a consistent interface, several architectural constraints are required to control the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.

The fifth constraint is the *Layered System*, which allows for multiple levels of abstraction within an architecture. This constraint prevents components from seeing beyond their immediate layer and forces them to interact with other components only through interfaces that are defined at that layer. By separating components into multiple layers, we can reduce the overall complexity of the system and promote substrate independence. Layers with encapsulated services allow new services to be layered on top of legacy clients without having to interface with them directly, simplifying components by moving infrequently used functionality to a shared intermediary. This intermediary can also be used to improve system scalability by dividing the load across multiple networks and processors, a task commonly referred to as load balancing.

The last constraint is the *Code-on-Demand*. This constraint is optional and specifies that a server can also deliver executable code to the client.

In summary, REST is an architectural style for developing loosely coupled applications that run over the network and

Fielding's proposal was to use the hypertext transfer protocol (HTTP) for inter-computer communication. Consequently, REST is based on HTTP. The REST architectural style uses the building blocks of HTTP to divide a system into "resources" based on unique URI patterns and uses the standard HTTP GET, POST, PUT, and DELETE to map operations on top of those resources. These standard HTTP verbs map to the familiar Create, Read, Write, Update and Delete (CRUD) operations used by generations of programmers.

B. API

The information from this section comes from the book "APIs: A Strategy Guide" by Jacobson, Brail, and Woods [1].

The application programming interface (API) is a means of providing access to data and services. It consists of a set of definitions and protocols through which, for example, two computer applications can communicate with each other over a network (especially the Internet), using a common language that both understand.

An API is also defined as a contract between providers and consumers. Once such a contract exists, developers are incentivized to use the API because they know they can rely on it. For developers, we refer to the people who use an API to create applications. The contract strengthens trust, which increases usage. The contract also makes the connection between provider and consumer more efficient because the interfaces are documented and consistent. The applications are then created based on this contract. To follow this contract, an API must follow a specification. Following a specification means, for example, that the API provider must describe exactly what features the API provides or when those features are available. It may also mean that the API provider must outline additional technical limitations within the API or additional legal or business limitations on using the API.

There are two types of APIs: private and public APIs. Public APIs are available to anyone without any contractual agreement (beyond agreeing to the terms of use) with the API provider. Private APIs, on the other hand, are used in a variety of ways, whether to support internal API efforts or for a partner to use that API.

C. REST API

The information from this section comes from the book "APIs: A Strategy Guide" by Jacobson, Brail, and Woods [1].

What makes an API a REST API depends on how closely it adopts the ideas (or constraints) of Fielding's dissertation. To enable easier development of an API that works on any machine or operating system, Fielding formalized the structure of the URI in REST, which uniquely refers to a resource or object or collection of objects. For example, a computer program on a machine wants to look at a list of customers. Due to the defined constraints of Fielding, it knows that there is a resource, defined by a URI, where it can get the list of customers. Through URI is possible to access to all the actions that can be performed on the entity "customers", like

deleting the list, or adding a new one. As originally proposed by Fielding, XML and hypertext are key parts of REST. In the age of modern computing, most platforms can talk to an HTTP server and a REST API requires only basic HTTP support.

There are several different kinds of REST architectures that are currently used for building APIs. Two of them are the *Pure REST* and the *Pragmatic REST*. Pure REST follows the dictates of Fielding's dissertation and a central concept to this style is the concept of "Hypermedia as the Engine of Application State" or *HATEOAS*. An API that follows this principle requires the client to discover the functionality that the API provides while using the API, rather than defining a list of things that the API can do in a static document. One advantage of this style is that clients and servers are able to grow and adapt as needed. The server can change the shape and functionality of an API without slowing down clients, because the client is built to adapt to server-side changes.

The second style, called *Pragmatic REST*, refers to a specific set of APIs that follow some key REST principles, but not all of them. These APIs are easy to use and understand, making them the overwhelming majority of public APIs. The reason for this is the *HATEOAS* principle's difficulty and complexity in its implementation from the developers' perspective. For example, a programmer who accidentally or intentionally hard-codes a URI path into an application may get an unpleasant surprise in the future, and the server-side API team may simply tell the client that their application has been rejected. Pragmatic REST focuses on making sure developers can accomplish their goals as quickly as possible and without writing a lot of extraneous code.

The choice between the two depends on the target audiences of the API, their possible approaches to developing applications for the API, and whether they prefer a more mature technology or one still in development.

D. Swagger

The information from this section comes from the book "Designing APIs with Swagger and OpenAPI" by Ponelat, and Rosenstock [11].

Swagger was initially just a user interface with some hints on how to write HTTP APIs in YAML files. As more tools were implemented to support the description of HTTP APIs, Swagger began to gain popularity. The rough guide they wrote gained attention and soon became a specification and a standard that many people and companies started to follow. In this way, all the tools and specifications that dealt with the description of HTTP APIs were collectively called "Swagger". Through this popularity, the specification became more mature and detailed, and was released as open source for the community to contribute to. In the end, also the big companies started to adopt this specification and used these tools. In 2015, the SmartBear³ company held the copyright for the term Swagger, which led to the renaming of the specification part from Swagger to OpenAPI specification. Nowadays, the

³SmartBear, <https://smartbear.com>

two terms Swagger and the OpenAPI specification are used interchangeably, but there is a big difference between the two. The first one is used to refer to the specific set of tools managed by the SmartBear company, which includes SwaggerUI, SwaggerEditor and many others. The second term refers to a description of HTTP-based APIs. This comes usually in the form of a YAML or JSON file. This file describes the inputs and outputs of an API, and we refer to this description as OpenAPI. The description of an API is not limited to the inputs and outputs, but can contain much more information, such as where the API is hosted, whether authorization is required, what kind of response a consumer might receive, etc. Definitions can be written with tools, by hand or generated from code. Once the API has been written, we refer to it as "described". Finally, when this API has been described in a definition, it becomes a platform for the tools that use it, hence the term OpenAPI, which refers to a standard description of RESTful APIs.

E. API Design Rules

To write clean and understandable APIs, implementing design patterns can be a useful tool to achieve these goals. In 2011, Massé published a book named "REST API Design Rulebook" [3]. In this book, he gives a very detailed description of RESTful API design rules. The goal of his work is the improvement of the quality of REST APIs while also providing a standard on certain mechanisms and specifications of RESTful Web service interfaces. These rules are divided into multiple categories such as URIs, Resources, HTTP request methods, responses, metadata design, and representation design. Each rule consists of a description, and often some examples for implementing patterns as well as anti-patterns. These rules lay the groundwork for our tool implementation and also the basis for a lot of related work we will look at in chapter III.

F. Natural Language Processing (NLP)

The information from this section comes from the book "Real-World Natural Language Processing: Practical Applications with Deep Learning" by Hagiwara [12].

Natural Language Processing (NLP) is a subfield of the large field of Artificial Intelligence (AI). In this field, the focus is on computational approaches to human language processing, understanding, and generation. In this field, there are many algorithms that can solve problems related to human-produced text. With these NLP algorithms, we can analyze a text-based input and generate information that helps to better understand the text without human intervention. Things like labels, semantic representations, and many others can be useful in understanding part of the input text without requiring a person to read the entire input manually.

The "natural" part of NLP is used in contrast to formal languages. Here, all the languages spoken by people today are considered "natural" because they have evolved naturally over time. In contrast, all languages that have strictly and unambiguously defined syntax and semantics are formal languages.

Examples of formal languages are programming languages such as Java, Python, and so on. In these languages, we always know whether something is grammatically correct or incorrect, and running a compiler or interpreter for these languages always results in a syntax error or not. In natural languages, the problem is more complicated because it is not always so easy to define what is grammatically right or wrong. Human languages are ambiguous, which means that interpretation is not always clear or unique.

Two well-known libraries used in this area are Apache OpenNLP and WEKA. Apache OpenNLP is an open source Java library used for natural language text processing. This library provides standard services such as tokenization, sentence segmentation, and parsing, which are used widely in the deep learning and NLP-tasks. The main advantages that make this library so popular are the already trained models that can be used and support a wide range of languages and only need to be imported to use them for the various functions already mentioned. Weka is another open source Java software that includes tools for data classification, regression, association rule mining, and visualization, and can be used to train new models based on categorized data.

III. RELATED WORK

Several different publications in the field of API design have the goal of making API definitions cleaner and more understandable. For this, several researchers have made suggestions or rules on how to write API definitions so that they appear clearer or easier to understand.

In the paper "Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach" [5], Palma et al. present an approach to detect (anti-)patterns in Rest APIs. For this, they manually analyzed REST patterns and anti-patterns to identify their characteristics. Out of those, they defined detection heuristics that were then used to implement detection algorithms for each (anti-) pattern. As some of the analyzed (anti-) patterns contained dynamic properties, a static as well as a dynamic analysis was done for 12 popular APIs. The result was an heuristic-based approach they call Service Oriented Detection for Antipatterns in REST (or short SODA-R) to detect (anti-) patterns in RESTful systems.

A year later in 2015, Palma et al. released the paper "Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti)Patterns" [6], in which they present an approach to detect linguistic (anti-)patterns. The goal of this approach is to make an API more understandable and maintainable. However, this linguistic analysis is only limited to contents of English dictionaries and therefore to the English language. Like in their previous work, their approach follows the manual step of analyzing (anti-) patterns with the help of literature and writing a detection algorithm based on the characteristics discovered in the analysis of the (anti-) pattern. To validate their work, they used 15 RESTful APIs.

Later, in 2017, Palma et al. released a paper called "Semantic Analysis of RESTful APIs for the Detection of Linguistic

Patterns and Antipatterns” [4] in which they present a new approach for semantic analysis of RESTful APIs (short: SARA) which employs syntactical as well as semantic analysis for the detection of linguistic (anti-) patterns. In their work, they provide 12 detailed definitions of linguistic (anti-) patterns and apply their detection algorithms on 18 widely-used RESTful APIs. Results of this application show an appearance of these antipatterns in the set of APIs, particularly in the form of poor documentation practices. Moreover, the new approach described in this work has a higher accuracy in detecting linguistic (anti-) patterns compared to the approach described in their previous work [6].

In 2017, Petrillo et al. studied in their work ”A Lexical and Semantical Analysis on REST Cloud Computing APIs” [13] the lexicons (terms used in their APIs) and linguistic (anti-)patterns of 16 REST Cloud Computing API providers. In their study, they discover that the lexicons of each provider are very different from each other and do not share many terms. Contributions of this work include an analysis tool called CloudLex to extract and analyze REST Cloud Computing lexicons, an analysis of the terms found in the 16 providers mentioned above, and an analysis of 23,000 URIs for linguistic (anti-)patterns. 54% of these URIs follow the Contextualized Resource Names pattern, but 62.82% contain the Non-pertinent Documentation antipattern. The tool they used is claimed to have a precision of 84.82%, a recall of 63.57%, and an F1-measure of 71.03%.

With their Work ”On Semantic Detection of Cloud API (Anti)Patterns” [14] published in 2018, Brabra et al. aimed to support cloud developers with enhancing their REST management APIs by providing a compliance evaluation of the Open Cloud Computing Interface (OCCI), REST best practices, and recommendation support to follow these principles. Although OCCI helps with the management of resources, it does not help with implementing best practices of REST design, resulting in reduced quality of REST definitions. To detect such (anti-) patterns, they propose a semantic approach which helps to define and detect such (anti-) patterns and offers suggestions that follow best practices for OCCI and REST APIs.

In their work ”Which RESTful API Design Rules Are Important and How Do They Improve Software Quality? A Delphi Study with Industry Experts” [7] published in 2021, Kotstein and Bogner conducted a Delphi study as a follow-up to previous studies that identified mismatches between theoretical RESTful concepts and practical implementation. For that, they confronted eight API experts and presented them 82 REST API rules originally designed by Massé [3]. They asked them to rate the importance of each rule as well as their impact on software quality. Out of the 82 rules, 37 were rated with low importance, 17 with medium importance, and 28 rules with high importance. The most important rule categories here were rules pertaining to the identifier design of URIs and the interaction design with HTTP. These make up around 55% of the total rules and around 50% of the rules classified by the experts with high importance.

In his M.Sc. thesis ”Haben Design-Regeln Einfluss auf

die Verständlichkeit von RESTful APIs? Ein kontrolliertes Experiment” [8] published in 2021, Pfaff conducted an experiment to analyze if implementing RESTful design rules has an impact on the understandability of RESTful APIs. In the experiment, 105 people were presented with different REST API fragments and were asked to grade their understandability. The experiment contained fragments complying with RESTful design rules as well as fragments violating these rules. The rules selected for this experiment were the rules which, according to the study conducted by Kotstein and Bogner [7], were supposed to have a high impact on the understandability. Results of this experiment revealed a significant negative impact on understandability when 11 of the 12 RESTful API design rules were violated. This effect was also independent of the experience of the participant with REST APIs.

In summary, several works have been published on implementing API design rules to improve quality, which are for the most part heavily inspired by the design book from Massé. While the earliest work of Palma et al. [5] and Brabra et al. [14] focused more on general implementation of rules for APIs, later works of Palma et al. and Petrillo et al. [13] focused more on the linguistic problems of API definitions. As a reaction to observed mismatches between theoretical RESTful concepts and practical implementations, Bogner and Kotstein conducted a Delphi study [7] to observe which rules from Massé would have the most impact on software quality in the eyes of industry experts. Their impact later was validated in another study by Pfaff, in which he concludes that these rules do have an impact on software quality attributes, such as the understandability of API definitions. We see our work as a follow-up to the above mentioned studies. In this work, we would like to present a prototype that implements the rules labeled with high importance by the experts in the study conducted by Kotstein and Bogner. While tools exist that check API definitions such as the tool Zally by the company Zalando, there is no publicly available tool to check most design rules analyzed in the Delphi study. Our goal is to provide a tool which offers static and dynamic analysis of API definitions to identify rule violations. Ideally, this would become a useful tool to support a user in writing API definitions of higher quality.

IV. TOOL IMPLEMENTATION

This chapter deals with the implementation of the tool-supported approach through a Command Line Interface (CLI). First, related tools are discussed and their advantages and disadvantages are highlighted. Afterwards, we describe the foundation on which the tool is implemented and how the design rules for OpenAPI definitions are realized.

A. Existing Tools for OpenAPI Analysis

Since there are already several open-source tools that address the quality of OpenAPI documents, these were investigated initially. Many of them additionally offer the possibility

to further develop rules. Therefore, some of these tools could have turned out to be a good foundation. As a starting point for the research of related validators, the list on the NORDIC API page⁴ was considered. However, several listed tools have not been maintained for some time, and are accordingly abandoned. Nonetheless, several companies show how seriously they take the analysis of OpenAPI documents. For instance, PayPal⁵, IBM⁶, and Zalando⁷ found it important enough to create and publish their own linters with documentation. The latter two tools in particular are well maintained and have great support, which is why we took a look on them. Table I shows a comparison between Zally and the IBM OpenAPI Validator and lists the main advantages and disadvantages of both tools.

IBM has implemented 67 rules for its OpenAPI Validator⁸. Thereby, the tool supports and extends the rule set of Spectral⁹, a linter of Stoplight¹⁰. JavaScript is used as the programming language, which suited us well, because it was already known in the team. Besides very good documentation for each rule, it is also well described how rules can be adapted or created. However, the design of the existing rules as well as how to develop new rules are focused on Linter activities. Therefore, the tool is well-suited for static and syntactic rules. As an example, many rules examine whether a field exists and whether it is non-empty. Unfortunately, existing rules are usually rarely configurable and have only a few options to choose from. Especially for the case convention, mostly only snake case or camel case are supported, which would contradict our analyzed rules. The fact that the tool focuses on pure static and syntactic rules made it unsuitable for us, just like its foundation Spectral.

On the other hand, Zalando's tool Zally¹¹ supports the examination of APIs against the rules defined in Zalando's RESTful Guidelines¹². The tool is strongly advertised to increase the quality of APIs and to provide best practices and guidance. Besides an easy-to-use CLI, an intuitive web app is also offered as an editor with feedback. Furthermore, a high degree of customization could be possible, and it could be easy to implement new rules. However, the documentation is sufficient but not as good as the one from the IBM validator. Moreover, unlike IBM's tool, Zally supports semantic rules such that resource names must be plural, which is a strong advantage. Nevertheless, we eventually decided against using Zally as the foundation for our development because we felt that it would not be possible to quickly learn the Kotlin programming language and become familiar with the framework of the tool. Since, as already described, the other tools are either not maintained or too much based on the static analysis, we decided in the team to implement the CLI from scratch.

B. CLI Implementation

Because the application was programmed from scratch, we decided to implement a CLI for the time being. The reasons for this were that it was possible to focus more on the important logic of the rule implementation instead of building a nice GUI on top of it. Furthermore, the target group has enough experience to use a CLI. As the CLI is not built on top of existing tools, we were free to choose the programming language and frameworks. The choice was not easy because each of us had different experiences with different programming languages and different ideas about how to implement such a tool. The proposed languages to use for the implementation were Java, Python, and Rust. Each of these languages has advantages and disadvantages, such as a pre-existing familiarity with one language or a difficult learning curve with the other language. To make a good choice, we decided to follow the studies already done in this direction and investigate which language and library are used to develop such a CLI. In the study by Bogner et al. [15] and the two studies by Palma et al. [5] [6], Java was used as the main language for developing such a tool. These studies led us to prefer Java. After investigating the possible frameworks and libraries we would need for the implementation of the rules, we learned about the Swagger parser and its library for Java¹³. Since parsing is an important aspect of our tool, we chose Java as the language for this project. We used Micronaut¹⁴, an open-source JVM-based framework for building applications with faster startup and lower memory requirements, as our main framework. We also used the picocli¹⁵ framework to create a CLI.

C. CLI Architecture

The goal of implementing the CLI was to automatically detect and report design rule violations and link them to software quality attributes. Therefore, based on the requirements derived from the analysis tools and the studies, an architecture of seven components was created. Figure 1 gives an overview of the components and how they work together.

- 1) **CLI Input/Output**: This component serves as an interface to the user. It asks for user input, such as a path to an OpenAPI document, as well as provides the output of a report file.
- 2) **Parser**: This component is responsible for parsing the input. As the core of this component we have the Swagger parser library, which can read URLs or *JSON/YAML* files directly and facilitates access to the data through already defined methods. After the parsing process is complete, a cleanup process is also performed on the resulting data. Since we do not need all the data, we filter out the data that is not needed for the violations.
- 3) **Rule Analyzer**: This module is the core element of the CLI. Here, the parsed OpenAPI object is passed

⁴OpenAPI Validator List, <https://nordicapis.com/8-openapi-linters/>

⁵PayPal, <https://www.paypal.com/de/home>

⁶IBM, <https://www.ibm.com/de-de>

⁷Zalando, <https://www.zalando.de/>

⁸IBM OpenAPI Validator, <https://github.com/IBM/openapi-validator>

⁹Spectral, <https://github.com/stoplightio/spectral>

¹⁰Stoplight, <https://stoplight.io/>

¹¹Zally, <https://github.com/zalando/zally>

¹²Zalando's RESTful Guidelines, <https://opensource.zalando.com/restful-api-guidelines/>

¹³Swagger Parser, <https://github.com/swagger-api/swagger-parser>

¹⁴Micronaut, <https://micronaut.io>

¹⁵picocli, <https://picocli.info>

TABLE I
PROS AND CONS FOR EXTENDING EXISTING OPENAPI VALIDATORS

	IBM OpenAPI Validator	Zally (Zalando)
Pros	<ul style="list-style-type: none"> - Large company with good maintenance and support - Possibility to implement new rules - Very good documentation for rule development - Supports rules from Spectral Linter - Programming language JavaScript known in the team 	<ul style="list-style-type: none"> - Large company with good maintenance and support - Highly customizable and possibility to implement new rules - Documentation for rule development
Cons	<ul style="list-style-type: none"> - Design and extensibility of rules geared towards Linter activities - Ideal for static and syntactic rule development - Insufficient configurability for most rules 	<ul style="list-style-type: none"> - Programming language Kotlin not known in the team - Too rigid and not adaptable in a short learning period

to the active rules and is analyzed. Information about the status of the rule is received from the `Config` component. If a violation occurs during the analysis, information is given to the `Violation` component, which returns a `Violation` object. After a successful analysis, the collected violation objects are passed to the `Report` module.

- 4) **Violation:** This module is designed to provide a unified rule violation. As input it gets information from the `Rule Analyzer` component and returns a `Violation` object. This contains: the rule object, the path of the violation, the line where the violation occurred, a description of the rule, and a improvement suggestion.
- 5) **LOC Mapper:** To provide the user with a better feedback on the location of the violation found, with respect to a path, some form of localization is required. For this purpose, each path in an OpenAPI definition is mapped to a line of code. All this is implemented by the `LOC Mapper` component, which takes an OpenAPI definition *JSON* or *YAML* as input and then performs a mapping process between the paths and the location in the file.
- 6) **Report:** This component creates two different type of reports. The first one is a Markdown file that is saved locally in the same folder where the analysis was performed. The second is a summary report that shows the user basic information about the detected violations and is displayed directly in the CLI. The information required to create a report comes from the `Rule Analyzer` component.
- 7) **Config:** This module is responsible for the overall configuration of the CLI. The configuration includes the selection of the active rules to be analyzed and the security scheme by the user input. This configuration is then used by the `Rule Analyzer` and `CLI Input/Output` component.

More about individual components is discussed in the next section IV-D about the CLI workflow.

D. CLI Workflow

The process of analyzing an OpenAPI definition is fundamentally rather simple. A very simplified representation of the steps can be seen in Figure 2. To achieve this workflow, we have programmed three main components:

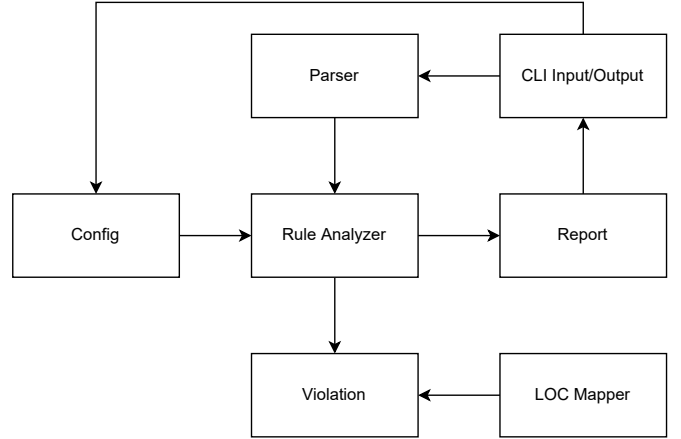


Fig. 1. CLI architecture

- 1) output component, for outputting text through the CLI, which is needed in step one and four in the analysis.
- 2) analysis component, that handles functions related to the analysis, which is used in the second step.
- 3) report component, for generating an overview of the violated rules, which is needed in step 3.

To understand how the individual components work, all steps are explained one at a time. In the first step, the user is prompted by the output component to specify a path to an OpenAPI document. The path can be local or can be provided by a URL. The file must be in either JSON or YAML format. In addition to the path, the user can optionally add more flags:

- 1) `expertMode`, to manage the active rules
- 2) `out`, to generate a report on the violations found and save it locally
- 3) `title`, to give the report a custom title

Afterwards, the user is asked if they want to perform a dynamic analysis or if they want to remain with the static one. If the path was entered correctly, the CLI moves on to the second step.

Here, the path is passed to the analysis component, which locates the OpenAPI document and translates it through the Swagger parser into a format we can process. It does not matter if the OpenAPI definition is in version 2.0, because it is first converted to a comparable OpenAPI 3.0 definition. Furthermore, the parser is very tolerant of violations of the schema, such as omitting content. The Swagger online

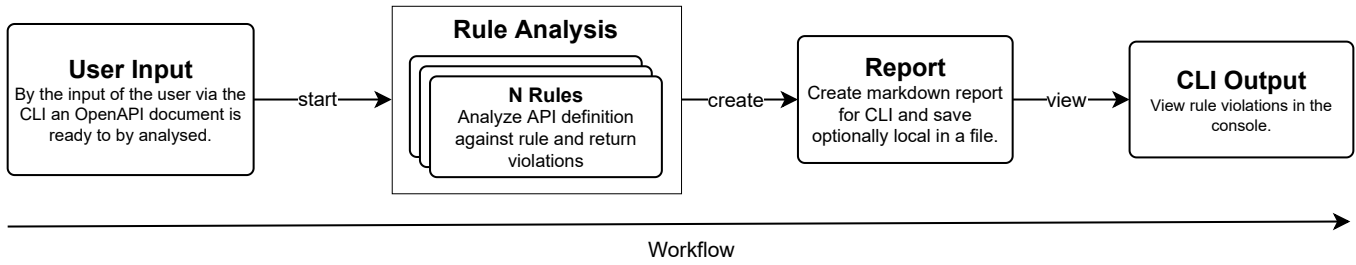


Fig. 2. An OpenAPI analysis workflow by the CLI

editor¹⁶, on the other hand, is much more restrictive and only parses a proper definition. The parsed definition is now ready to be analyzed. The goal of the analysis is to detect and report rule violations. Since in the implementation the structure of these violations looks the same for each rule, we have created an interface for it. When creating a rule, the interface is then implemented and we thus make sure that we have all the necessary data for a report. Figure 3 represents this interface with the methods to be implemented. Besides a title, the category, the severity, the type (static and/or dynamic implementation), and the software quality attributes, the status of the rule is needed. These are the properties of a rule and are needed for the report. The backbone of the rule analysis is the `checkViolation()` method, which examines the parsed OpenAPI and returns the violations. More information about the implementation of the rules can be found in section IV-H. All active rules are then executed one after another and the rule violations are saved.

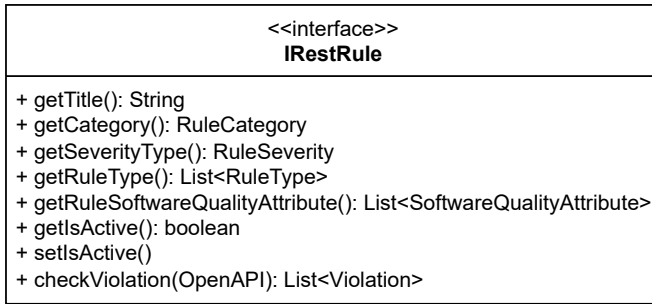


Fig. 3. Rule Interface

Subsequently, in the third step, the rule violations are passed to the report component, which generates two different Markdown outputs using a Markdown generator¹⁷. One output is directly for the console, which shows only the rule that was violated, as well as the corresponding path, and in which line it can be found, plus an optional second report for writing the violations to a local file, where additionally the category, the severity, the type, the software quality attributes, and an improvement suggestion is added. For a better user experience,

a progress bar for each rule has been implemented, which shows the percentage of paths analyzed so far.

As the last step, the report is displayed to the user in the console and, depending on whether the `out` flag was set in step one for generating a report, it is also saved locally. The file location will then be displayed on the console.

E. Dynamic analysis

During dynamic analysis, software is actually executed, i.e., in our case, requests are made to a RESTful API. Making such a request has two prerequisites that are not necessary for static analysis. First, the server of the RESTful API must be reachable. Therefore, the path to the server needs to be extracted and pinged. If it is reachable, the user input can be continued. The OpenAPI document is then searched for an already defined security schema. If one exists, the user is asked for the corresponding token. If none is available, a list of schemes is provided for selection. So far, there is the possibility to choose between `basic`, `bearer`, and `api-key`. Finally, the user has the option to save the schemas locally in a `config.properties` or to use them only once for the next analysis. If the tokens are saved, they are directly suggested to be used or replaced during the next analysis.

The focus of the dynamic analysis was to make sure that no further input is required in addition to the input of the tokens for the API. However, due to lack of time, dynamic analysis has been implemented for only one rule so far (see "Unauthorized" rule in Section IV-H). Nevertheless, some groundwork has already been done, but more work is needed to analyze other rules dynamically. Especially the generation of parameters and files needs some preconditions. Furthermore, no resources should be updated or deleted.

F. Rule customization and extension

Since the project was very time-limited, care was taken to make sure that the tool is easily expandable. This applies to the adaptation of the rules as well as to the creation of new rules. Since the Java code for the CLI is freely accessible, it is no problem to customize the functionality of the rules. For each rule, a single Java class was created, which can be found in `./rest-studentproject/src/main/java/rest/studentproject/rule/rules`. It is just as easy to implement a new rule. For implementing a new rule, it is merely necessary to create a Java class in the folder just

¹⁶Swagger Editor, <https://editor.swagger.io/>

¹⁷Java Markdown Generator, <https://github.com/Steppschuh/Java-Markdown-Generator>

mentioned, which implements the `IRestRule` interface. Then, a constructor with an `isActive` boolean is needed. Now the rule is automatically recognized and listed in the CLI. This is the minimum that needs to be done to implement a new static rule. However, for a dynamic analysis, it must first be checked whether it is requested by the user. Therefore, it is necessary to check the attributes `dynamicAnalysis` (boolean) and `securitySchemas` of the calling class. This is possible because they are public static in the Java class. The latter attribute is a list of user-specified security schemes for the API, either a bearer token, basic authentication, or an `api-key`. Based on this, the necessary information for a dynamic analysis is available and requests can be sent to the API.

For a better user experience, the progress bar can be called from the output class (`./rest-studentproject/src/main/java/rest/studentproject/rule/utility/Output.java`).

G. Development

For the development of the CLI, we used some form of prototyping [9]. The goal was to quickly have an executable software with a minimum of functions, which then serves as a foundation to request feedback from the stakeholders. Based on this feedback, the tool was then extended and kept executable. In addition to regular meetings with the stakeholders every two weeks, the team met once a week on-site to develop together, validate the tool, and address problems. Furthermore, the team did not know each other for long, so the face-to-face meetings contributed to a good working atmosphere, which improved motivation and productivity [16]. Moreover, working remotely could have some disadvantages, such as that better organization skills are necessary for the work, as well as the risk of losing contact with team members [17]. Further, pair programming was possible on-site, which positively affected the development of highly complex programming tasks, saved time on easier tasks, and helped in solving tasks that would be difficult or even impossible for one person [18].

GitHub¹⁸ was used to host the code, as well as to create and organize issues. Templates were used for the issues, as well as the pull requests, to promote description and completeness. Additionally, a pipeline was set up with GitHub Actions¹⁹ to build the CLI, run the tests, and perform security scans on the code through semgrep²⁰.

For the implementation of the tool, rules from Massé's book [3] were analyzed one by one. How the rules were selected is described in the next section IV-H. Afterwards, for the implementation of the individual rules, five steps were conducted:

- 1) a team member, who does not implement the rule, created the gold standard (more info in chapter V).
- 2) a rule was added to the CLI by implementing the interface.

- 3) tests were written by the developer of the rule.
- 4) a Markdown file was created for the implemented rule, which contains rule properties, a rule description, and implementation details.
- 5) a pull request was created on GitHub, which was validated by a third team member.

The third and fifth steps are particularly good for validating the tool, as the code has been reviewed by the developer and additionally by other team members. By constantly creating pull requests per rule, as well as through the pipeline via GitHub Actions, continuous integration was encouraged. Since there are several possibilities for implementing a design rule, and the implementation usually does not cover the complete scope of the rule definition, the documentation of the rule in the fourth step is important. Furthermore, documentation helps to extend and adapt individual rules.

H. Implemented rules

The rules implemented and analyzed in this paper are from Masse's book [3]. Deciding which rule to implement and which not was a long process that was discussed internally, and papers such as that by Kotstein and Bogner [7] and by Pfaff [8] contributed to the selection of the rule we ended up implementing. The selection process can be brought down into four phases. In the first phase, we focused on understanding the definition of the different rules, e.g., what to look for and what possible mistakes could lead to a rule violation. To speed up the process, we decided to use the categorization already created by Kotstein and Bogner. We started with the rules that were marked with "high importance" and assigned 6 rules to each of us to examine. In the second phase, when we had the necessary information, we proceeded to categorize the rules into three categories regarding difficulty of implementation: easy, medium, and difficult. Matching the implementation idea, the type of analysis was added: static and/or dynamic. Furthermore, the linking of the rules to different software quality attributes from Kotstein and Bogner were used and added to the rule. Quality attributes are: functional suitability, performance efficiency, compatibility, usability, reliability, maintainability and portability. We also categorized the rules according to the severity: Warning (may be), Error (should be), and Critical (must be). The severity indicates the urgency of a violation to be fixed. Naming conventions were inspired by IBM²¹. The tag `Critical` is derived from Massé's definition of the rule, in which "*must be*" is used as a phrase. It means that there is a critical condition with this violation and must be fixed immediately. An `Error` is assigned if "*should be*" is used as a phrase in the rule definition and means that there is an error condition with the violation, which should be removed. Lastly, the tag `Warning` is used when "*may be*" is used as a phrase and indicates a possible problem that may be fixed sooner or later. Understanding and categorizing the different rules made it easier to decide which rules could be

¹⁸GitHub, <https://github.com/>

¹⁹GitHub actions, <https://github.com/features/actions>

²⁰Semgrep, <https://semgrep.dev/>

²¹IBM Message severity tags, https://www.ibm.com/docs/en/ess/6.0.1_ent?topic=messages-message-severity-tags

implemented and which could not. This decision was made in the third phase. Then, in the final phase, some changes were made to the interpretation of the rules, if necessary, to enable implementation, and extensive testing was performed to guarantee that the rule was implemented correctly.

In the end, a total of 14 rules were implemented, which are shown in Table II. First, a brief background for the rules is given, and then each rule is described.

Many of the rules we implemented work with a part of the URI segment for an HTTP request. URI stands for **Uniform Resource Identifier** and is a sequence of characters that identifies an abstract or physical resource. According to RFC 3986²² a URI has following structure:

URI: scheme "://" authority "/" path ["?" query] ["#" fragment]

A possible example of a valid URI, following above structure would be:

`https://example.com:8080/shelter/animals?name=cat#rescue`

with "https" being the scheme, "example.com:8080" the authority, "shelter/animals" the path, "?name=cat" the query, and "#rescue" the fragment.

For the implementation of rules related to the URI, we work on the "path"-portion of the URI. In an OpenAPI definition of an API, the path portion can be found in the "paths"-segment. Each path in the "paths" segment may optionally contain a list of parameter definitions for both path variables as also query parameters next to the required definition of the path. It may also contain request bodies and response definitions for numerous HTTP response codes. In these responses, a detailed description of what kind of response object, such as their data type, will be received for each HTTP response code.

Rule: Forward slash separator (/) must be used to indicate a hierarchical relationship

"The forward slash (/) character is used in the path portion of the URI to indicate a hierarchical relationship between resources." – Massé [3]

The way we understand the rule written by Massé is (1) a forward slash (/) is to be used as means to separate words in the path of a URI and (2) the words separated by the separator have to be ordered hierarchically.

From the above definition of the URI by RFC 3986 we know that a '?' character defines the beginning of a query and the '#' character defines the beginning of a fragment. These characters are therefore illegal in the path segment, as they start new segments in the URI. Our implementation of this rule will as a result immediately flag a path containing either of those two characters as a violation of this rule.

Another class of violations is the use of a different character as a separator instead of the forward slash (/) defined as the separator by RFC 3986. By further

analysis we discovered that not only could a different character be used to replace the forward slash (/) (e.g., `.shelter.animals.cat`) but could also only replace a subset of the forward slashes (/) in the path (e.g. `/germany/states/baden-wuerttemberg/shelter-types.animal-shelter.animals.cat`). To capture all those cases we analyze the path and search for potential characters used as a separator with the help of complex regular expressions. The characters we look for in our implementation are: ',', ':', ';', '\', '/', '-', '=', ' '.

There are cases we deem to be violations of this rule but have not been implemented. The first case of a violation we do not cover in our implementation is the case where separators might be included in the values of path variables of the path. As these values cannot be identified through a static analysis of the paths: complex, resource-intensive dynamic analysis would be required. The second case we do not cover is hierarchy mismatches between the words used into a path, separated by a separator. E.g `/countries/germany/states/baden-wuerttemberg` would be a valid hierarchy while `/house/resident/apartments` would be an invalid hierarchy as it would make more sense to have "apartments" before "resident" as a house can have multiple apartments with an apartment having a resident. The quality attribute that is linked to this rule is maintainability and has a severity level of critical.

Rule: A trailing forward slash (/) should not be included in URIs

"As the last character within a URI's path, a forward slash (/) adds no semantic value and may cause confusion. REST APIs should not expect a trailing slash and should not include them in the links that they provide to clients." – Massé [3]

The way we understand this rule is that no forward slash (/) is to be included from the end of a path segment in a URI. To enforce this rule, we check each path segment in an OpenAPI definition if it ends with a forward slash (/) and in the case of a trailing forward slash (/), throw a violation.

The software quality attribute that is linked to this rule is maintainability and has severity level error.

Rule: GET and POST must not be used to tunnel other request methods

"Tunneling refers to any abuse of HTTP that masks or misrepresents a message's intent and undermines the protocol's transparency. A REST API must not compromise its design by misusing HTTP's request methods in an effort to accommodate clients with limited HTTP vocabulary." – Massé [3]

The way we understand this rule by Massé is that GET and POST requests may only be used for the use cases they are intended for. That means that GET requests should only be used to retrieve a representation of a resource²³ using the request URI. GET requests should not be used to create, modify, delete data, or even be used to start processes. POST requests, on the other hand, should only be used to create a new resource. It can also be used to send a request to a

²²Fielding et al. RFC 3986, <https://www.rfc-editor.org/rfc/rfc3986>

²³RFC 2616, <https://www.rfc-editor.org/rfc/rfc2616.html>

TABLE II
IMPLEMENTED RULES

Implemented Rule	Abbreviation
Forward slash separator (/) must be used to indicate a hierarchical relationship	Forward slash
A trailing forward slash (/) should not be included in URIs	Trailing slash
GET and POST must not be used to tunnel other request methods	Tunnel
GET must be used to retrieve a representation of a resource	GET resource
File extensions should not be included in URIs	File extensions
Content-Type must be used	Content type
CRUD function names should not be used in URIs	CRUD function names
Underscores (_) should not be used in URI	Underscores
401 (Unauthorized) must be used when there is a problem with the client's credentials	Unauthorized
Hyphens (-) should be used to improve the readability of URIs	Hyphens
Lowercase letters should be preferred in URI paths	Lowercase
A plural noun should be used for collection or store names	Plural noun
A singular noun should be used for document names	Singular noun
A verb or verb phrase should be used for controller names	Verb for controller

controller or any other data-handling process. POST requests should not be used to retrieve a representation of a resource, nor used to delete data. If a request does something it is not supposed to do, like deleting data through a GET request, we refer to this as tunneling. In this example case, a GET request is tunneling a DELETE request.

In our implementation, we considered two sources to possibly give us signs of a tunneled request. First, we consider the path and look if some kind of *CRUD* (Create Read Update Delete) operation is present. If a *CRUD* operation is found in the path segment, the operation is compared to the request type and checked for a match. In case of a mismatch, a violation is raised. Secondly, we check the description field of the path segment, if specified in the OpenAPI definition. Here, we search for keywords that imply a *CRUD* operation and compare it to the request type. If there is a mismatch, a warning is raised. We would like to note that this implementation does not capture all possible violation cases. For example, this implementation does not do any dynamic analysis of the path definitions. As a result, it is not checked if the server that receives the request, handles the request as specified in its use cases. Therefore, there is no guarantee the description of a path with a request type matches what happens on the server side.

To improve the analysis of this rule, we decided to also investigate the description and summary of a path. Normally, when writing a path, the description or summary refers to the purpose of the path. The definition of a delete request requires a corresponding description or summary that specifies how the action is performed or what the purpose of this path is. With this information, assumptions can be made about the type of request. For example, if the description is about deleting some elements in the database, but the request is of type POST, there is a discrepancy between the intent and the actual execution of the request. This problem could lead to tunneling requests. To identify this type of discrepancy, we implemented and trained a model using a Java library, namely Weka²⁴. To implement this model, we used the *Naive Bayes Multinomial*

classifier²⁵. This classifier was chosen because of the size of the vocabulary we have to work with. The vocabulary of a description or summary of a path can vary from small to large, and to achieve good performance and accuracy, the *Naive Bayes Multinomial* classifier is more suitable compared to other classifiers [19]. The model was trained using 3,227 manually categorized samples and achieved 89% accuracy. The data was categorized according to different types of requests and an additional category, that of invalid requests. A request is invalid if it attempts to perform more than one action and is not of type POST. Compatibility, functional suitability, maintainability, and usability are all software quality attributes that are linked to this rule and has the severity level of critical.

Rule: GET must be used to retrieve a representation of a resource

"REST API client uses the GET method in a request message to retrieve the state of a resource, in some representational form. A client's GET request message may contain headers but no body. The architecture of the Web relies heavily on the nature of the GET method. Clients count on being able to repeat GET requests without causing side effects. Caches depend on the ability to serve cached representations without contacting the origin server." – Massé [3]

The way we understand this rule by Massé is (1) GET requests should never contain a request body and (2) a GET request should always receive some kind of object that represents a resource in the case of an HTTP 200 response. To check for this rule in our implementation, we search within each path object in the OpenAPI definition of an API for GET requests. For all found GET requests, we check if they contain a request body. If they do, our implementation will raise a violation of the rule. If no request body is found, the defined responses in the path object is checked. Here, we look if either an HTTP 200 or a default response is present. If none are defined, a violation of this rule is raised. Additionally, if definitions of a default and/or HTTP 200 response exist, they are checked if a representation of a resource is defined.

²⁴Weka, <https://waikato.github.io/weka-site/index.html>

²⁵Naive Bayes Multinomial Classifier, <https://javadoc.io/static/nz.ac.waikato.cms.weka/weka-stable/3.8.4/weka/classifiers/bayes/NaiveBayesMultinomial.html>

For that, we check if a content type is defined inside the response or if a reference to an object specification exists. For re-usability purposes, OpenAPI allows objects to be defined in the components segment of the specification, so these can be reused for multiple requests and responses. If none of the above are defined, a violation of this rule is raised. This implementation also does not cover several violation cases. For instance, we do not check if a server returns the specified object from the OpenAPI definition. Additionally, it is not checked by this rule if a different request operator is used to retrieve a representation of a resource. These set violation cases are partly covered by the implementation for tunneling of GET and POST requests. The software quality attributes compatibility, functional suitability, maintainability, and usability are all linked to this rule. this rule has a severity level of critical.

Rule: File extensions should not be included in URIs

"On the Web, the period (.) character is commonly used to separate the file name and extension portions of a URI. A REST API should not include artificial file extensions in URIs to indicate the format of a message's entity body. Instead, they should rely on the media type, as communicated through the Content-Type header, to determine how to process the body's content." – Massé [3]

For this rule, only the first part of the definition was implemented. However, the second part is covered by the following rule "Content-Type must be used". The first step in the static analysis is to divide the path into segments. Then, to check if a segment ends with a dot and a file extension (.file-extension), a list²⁶ of 838 different file extensions is searched. If one of them matches the end of the segment, a rule violation is returned. This list can be extended very easily if file extensions are missed. The list is located in the folder: Projektarbeit-Master/rest-studentproject/src/main/java/rest/studentproject/docs/file_extensions.txt.

Besides a severity of an error, the rule has an impact on maintainability.

Rule: Content-Type must be used

"The Content-Type header names the type of data found within a request or response message's body. The value of this header is a specially formatted text string known as a media type [...]. Clients and servers rely on this header's value to tell them how to process the sequence of bytes in a message's body." – Massé [3]

In this rule, the responses and the request bodies are examined. It is examined whether a content type was specified directly or in the components. If the component references again to another component, it is also returned a rule violation because otherwise, it would be necessary to search for the content type. For the response 204 (No Content), no violation is given if no content type was defined.

However, the content-type of the path parameters

was not examined. The fact that this was forgotten during implementation became apparent during the evaluation of the Gold Standard and thus, fixed later.

The severity level of the rule is set to critical, and there may occur usability and compatibility issues when the rule is violated.

Rule: CRUD function names should not be used in URIs
"URIs should not be used to indicate that a CRUD function is performed. URIs should be used to uniquely identify resources [...]". Therefore, *"HTTP request methods should be used to indicate which CRUD function is performed."* – Massé [3]

For this rule, the path was divided into its individual segments. Afterwards, the segments were statically examined to see whether they contain the direct keywords of the CRUD operations (get, post, delete, put, create, read, update, patch, insert, select). Since these keywords are also often partial words of others, it was first checked whether this is the case. 605 words are checked, which have the above mentioned keywords as substring, e.g. *postbox*, *target*, etc.. If the keywords are not a substring of the word list, a violation is returned.

However, the list of CRUD operations was extended by synonyms after the evaluation of the Gold standard: *add*, *retrieve*, *fetch*, *purge*. Thus, the list of words containing these operations as substring was increased to 833.

If the rule is violated, there is a risk of lower maintainability and usability; thus, these violations should be removed (severity error).

Rule: Underscores (_) should not be used in URIs

"Text viewer applications (browsers, editors, etc.) often underline URIs to provide a visual cue that they are clickable. Depending on the application's font, the underscore (_) character can either get partially obscured or completely hidden by this underlining. To avoid this confusion, use hyphens (-) instead of underscores." – Massé [3]

For this rule, the whole path is searched for an underscore. If one is found, a violation is returned. Violations should be removed as they affect the maintainability.

Rule: 401 (Unauthorized) must be used when there is a problem with the client's credentials

"A 401 error response indicates that the client tried to operate on a protected resource without providing the proper authorization. It may have provided the wrong credentials or none at all." – Massé [3]

To investigate whether a 401 response was defined for a path in an OpenAPI definition, a static and dynamic analysis was implemented. First, the static analysis is described: In an OpenAPI definition, the security scheme is specified either globally for all paths, or locally for one path. Thus, when a security scheme is given on a path, it is examined whether a 401 (Unauthorized) response has been defined. This should be given if problems occur during authorization. If this response is not given, a violation is returned. Now, the dynamic analysis is described: If no violation was given by the static analysis, a request is sent to the API. For this, the security schema entered by the user is used and modified so that it is no longer correct. If no 401 response is returned, a

²⁶List of filename extensions, https://en.wikipedia.org/wiki/List_of_filename_extensions

violation is thrown because no authorization was allowed due to the falsified tokens. So far, this only works for GET requests because otherwise resources could be modified; e.g. if the user specifies a security scheme, but there is no authorization at the API. This was no longer improved due to time constraints.

Currently, it is also not investigated whether a wrong status code is used in case of problems with the credentials. For future development, an AI could be trained by the descriptions in the responses, as it was done with the tunneling, which then indicates whether the correct response for problems with the authorization has been used.

The rule has a critical severity, and has quality characteristics regarding compatibility, maintainability, and usability.

Rule: Hyphens (-) should be used to improve the readability of URIs

"To make the URIs easy for people to scan and interpret, use the hyphen (-) character to improve the readability of names in long path segments." – Massé [3]

To improve readability, the words that make up a sentence should be clear and distinct. When writing a path for an OpenAPI definition, it may happen that two or more words are strung together without using a readable separator. To implement this rule, we first divide the entire path into path segments and then check for each path segment to see if there are more than two words that make up the path segment. If two or more words make up the path segment, there is a violation. To determine if something is a word, we used a dictionary of 126k English words provided by Wordninja²⁷. If a second word is found after the first one is detected and no hyphen was used as a separator, then the tool will create a violation for the path.

The software quality attribute that is linked to this rule is maintainability and the severity is that of an error.. **Rule: Lowercase letters should be preferred in URI paths**

"When convenient, lowercase letters are preferred in URI paths since capital letters can sometimes cause problems." – Massé [3]

This rule checks if a given path contains an uppercase letter, and if it does, then there is a violation. The implementation is static, and the logic is straightforward.

The software quality attribute that is linked to this rule is maintainability and the severity is that of an error.

Rule: A plural noun should be used for collection names or store names

"A URI identifying a collection should be named with a plural noun, or noun phrase, path segment. A collection's name should be chosen to reflect what it uniformly contains. A URI identifying a store of resources should be named with a plural noun, or noun phrase, as its path segment." – Massé [3]

We have implemented the syntactic part of this rule with some deviations compared to the original rule described by Massé. The idea behind this implementation is that a collection must be followed by a document name that is not plural. For example, a path can be formed as fol-

lows: collection/document-name/collection/document-name... or document-name/collection/document-name... but if we have two document names one after the other, we have a violation of this rule. This rule is implemented statically, where each path segment is checked and assigned a type, singular or plural. The plural stands for a collection. The singular for a document name. The assignment to the respective type is made possible with the help of the OpenNLP library provided by Apache²⁸.

The software quality attribute that is linked to this rule is maintainability and usability. The severity of this rule is that of an error.

Rule: A singular noun should be used for document names
"A URI representing a document resource should be named with a singular noun or noun phrase path segment." – Massé [3]

The implementation of this rule is similar to the plural form, but the violation occurs when there are two collections in sequence, e.g., collection/collection/document-name. This rule is also implemented statically using the OpenNLP library.

The software quality attribute that is linked to this rule is maintainability and usability. The severity of this rule is that of an error.

Rule: A verb or verb phrase should be used for controller names

"Like a computer program's function, a URI identifying a controller resource should be named to indicate its action." – Massé [3]

It is not so trivial to trace a noun to a verb form. This problem could lead to many false-positives if the user did not want to use a controller resource for a particular path, but only a noun at the end of the path. To implement this rule without generating false-positives, we check whether the request is of type GET or POST and whether the last path segment is a verb. If this is the case, there is no violation, but if the request is of a different type and the last path segment contains a verb, there is a violation. This rule is implemented in a static way, where we split a given path into path segments and analyze the last one. The rule requires using a verb to indicate controller actions. A controller can only be of type GET or POST.

The software quality attribute that is linked to this rule is maintainability and usability. The severity of this rule is that of an error.

V. METHODOLOGY

The described implementation of the tool support for the detection of design rule violations in OpenAPI documents has to be evaluated. Thus, to make a statement, a large sample size is needed to be tested by the tool, which is described in the study object section. Afterwards, the data collection is discussed and which metrics we used to measure robustness, effectiveness and performance.

²⁷Wordninja, <https://github.com/keredson/wordninja>

²⁸OpenNLP, <https://opennlp.apache.org>

A. Study Objects

After doing some research, we discovered an online repository for API definitions "apis.guru"²⁹ that followed the OpenAPI specification. With the help of a script, we mined all OpenAPI definitions. Furthermore, the script saved these OpenAPI definitions as JSON files locally to avoid deletion online. This resulted in a list of 2,346 distinct OpenAPI definitions. These definitions heavily vary in size and cover a wide range from official APIs from big companies like Amazon and Microsoft to museums and governments. It is notable that in all the OpenAPI definitions we mined, there were barely any multiple versions of the same API. We did notice that description fields from the same developer would occasionally share the same description field content. In total we found 547 API definitions that shared the same description with another API definition. There are notable differences between all mined APIs: Some use many references to the components to define their definitions, while others barely use them at all. Some would define responses for multiple HTTP responses, while others would only have the HTTP 200 response and sometimes also a default case to catch all other HTTP codes. This kind of variance in the OpenAPI definitions collected makes them an ideal testing ground for our implementation of API design rules. For our testing, we only did a static analysis of the OpenAPI definitions and skipped the dynamic analysis. The reason is that the security inputs have to be provided manually, making it impossible to provide them for all APIs. Many APIs require registration outside of the API or it was unclear about how to register at all.

B. Data Collection

To examine how well our implementation of the rules by Massé [3] perform, we decided to test the tool under three sets of criteria. These three sets of criteria can be summed up by the terms *robustness*, *effectiveness*, and *performance efficiency*. The following describes the three metrics and how the data was collected.

1) *Robustness*: Under "robustness" we understand how well our implementation analyzes OpenAPI documents without producing an error. To make any meaningful statement about the state of our implementation in terms of robustness, we need a large sample size of OpenAPI definitions. Therefore, we took all 2,346 OpenAPI definitions we have collected and analyzed them. For that, we wrote a script that would automatically run our implementation with each OpenAPI definition as input. With a second script, we created a list of all OpenAPI specifications that failed to be processed, and, as a result, did not generate a valid output file. For that, the script checked the output folder of our implementation and compared reports in the folder with the compiled list of OpenAPI definitions. It took the difference between these two sets and created a list of all OpenAPI specifications with a missing report file. This marks the first iteration. In the second

iteration, we re-run those failed specifications and checked the error messages that were created, and categorized them by their errors.

2) *Effectiveness*: In the static analysis of OpenAPI documents, the definitions are analyzed without a request being made. Thus, potential rule violations can be detected quite early and quickly fixed. However, when analyzing OpenAPI documents, tools are not always completely effective. There may be issues, where rule violations are incorrectly detected, or they may not be detected at all. Therefore, it is necessary, to measure the effectiveness of this tool with two metrics – recall, and precision. To define these metrics, we use the following concepts:

- A *true-positive* is a genuine rule violation, which is successfully detected and reported by the tool. Henceforth, the number of true-positives will be abbreviated by TP.
- In case of a *false-positive*, the tool shows a rule violation even though there is no violation in the document. The abbreviation for the number of false-positives is now FP.
- A *false-negative* is a rule violation in the OpenAPI definition, which was not recognized as such by the tool. The number of these are now indicated by FN.
- Lastly, we have a *true-negative* if there is no rule violation and the tool does not report a violation. TN is the number of these [20].

To indicate the quality of our tool regarding effectiveness, we calculate recall and precision. They are calculated by ratios between true-positives, false-positives, and false-negatives. **Recall** is the percentage of existing rule violations that are correctly identified in a document. The percentage is calculated by:

$$Recall = \frac{TP}{TP + FN} \quad (1)$$

Precision, on the other hand, is the percentage of rule violations reported by the tool that are actually correct. The percentage is calculated by:

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

A perfect tool would output a value of 1 for recall and precision; i.e., no false-positives and no false-negatives. However, a perfect analysis is usually very difficult and impossible to achieve. Often, tools come close to optimal values. Mostly, they are then either under-approximated and therefore complete because they report no false-positives but a high number of false-negatives. Thus, a high number of faults are not detected, while the faults that are detected are genuine faults. Alternatively, they may be over-approximated and thus have few false-negatives but a high number of false-positives. This means that almost all faults are detected, but many incorrectly classified faults are also detected. The goal should therefore be to have as few false negatives and false positives as possible; i.e., to miss only some faults and at the same time not misclassify too many faults.

²⁹OpenAPI repository, <https://apis.guru/>

Normally, recall is difficult to measure in practice because it is hard to identify violations that are not detected. However, to measure the recall, i.e. the true-positives and false-negatives, we asked seven software professionals familiar with REST to define a gold standard for our implemented rules and thus violations for the rule set. If possible, they should provide edge cases so that false-negatives could be identified. Afterwards, for isolation, an OpenAPI definition was created for each rule, in which the experts violations were inserted, i.e. there was a fixed number of violations by the experts regarding a rule in a definition. After the tool finished the analysis, the number of false-negatives could be calculated per rule, namely the number of violations in the OpenAPI definition minus the number of correctly detected violations (true-positives) by the CLI. In addition to the rule violations from external experts, a gold standard was created by developers. For this, rule violations were defined even before implementation. Care was taken to make sure that the violations created were not written by the same developer as the one who implemented the rules. However, since we developers were all at about the same level of knowledge and had not yet gained much experience with OpenAPIs, these violations were rather straightforward and mostly included in the violations defined by the experts. That is why we merged the two gold standards. If the experts did not consider a violation that we defined in our standard, we added it to theirs. To have a comparison to other tools, we then let Zalando's tool Zally analyze the gold standard. However, since we have only implemented four identical rules, only these will be compared.

Precision is easier to measure in practice by taking a specific set of violations and checking them to see if they were correctly classified by the tool. To obtain a large set, all three developers validated a set of the received violations from the robustness analysis. A Python script was created that randomly assigned 30 different OpenAPI violation reports to each of us. As some of these reports contained many violations, a limit of 30 violations per report was set for the investigation. These 30 violations were also randomly selected by a Python script. The title of the OpenAPI, the violation examined, the associated path and whether it was a false-positive or true-positive was recorded in an Excel file. Finally, we could specify the precision for almost every rule; however, there were also the trailing slash and the verb for controller rule, which did not occur so often and were therefore not included into the manual analysis.

3) *Performance efficiency*: When performing an analysis, whether of code or other functions, the user expects a relatively fast processing time and the ability to use the tool without consuming too many resources. These two aspects are part of the performance efficiency of ISO 25010³⁰, which describes the quality models for software products. The performance efficiency model represents the performance of a tool based on the resources, time, and capacity used. In our study, we

focused on the resource usage and time aspect. An interesting study by Franca and Soares [21] shows that performance efficiency is an important quality characteristic, especially in applications based on service-oriented architecture. Time-behavior analyzes the response time during the execution of a given function. In our tool, we considered the time it takes to analyze a file. As for resource use, which focuses on the amount of resources used, we considered the RAM and CPU use required to analyze a file.

To conduct an experiment and collect representative data for an exploratory study, we measured the respective CPU and RAM usage for each analysis of the 2,331 files, as well as the time it took to analyze the violations of an OpenAPI. To make the best decision on which tool to use and how to perform the measurements, we oriented ourselves to the paper "A unifying approach to performance analysis in the Java environment" by Alexander et al. [22]. In this paper, it is discussed how to prepare a performance analysis, which objectives are involved in the analysis, and what factors are important for the application and analysis. This description allowed us to get an overview of the available tools and decide which one was better suited for our analysis. For RAM and CPU usage, we used the Python library psutil³¹, which has more than 8k stars on GitHub and is supported by a large community. This library retrieves information about running processes and system load. In this way, we analyzed only the Java process responsible for the analysis and not the Python script or other process. Then, we wrote the data to a CSV file. For the time it takes to run and complete an analysis, we used the `time` module provided by Python. We measured the time it takes to run the Java CLI to analyze a file, which is just the execution of the jar file with the file as a parameter. Then, we subtracted the start time from the end time to obtain the time needed in seconds, and the data was also written to a CSV file. To make sure that the analysis and measurement went smoothly, we compared the generated reports with the files of the measurement data created in this way. All reports and CSV files had the same name as the analyzed files, so it was easier to compare if and which files were missing.

An analysis of all files without further classification would give little information about the overall performance of the tool. Each file is different from the other. There are large files that could take a lot of time to analyze, which would distort the result of this analysis. For this reason, we decided to categorize each file based on the number of paths. In this way, it is possible to gather much more detailed information about which category works well and which could be a problem. In total, five categories were created based on the number of paths: *Very low*, *Low*, *Medium*, *High*, and *Very high*. This categorization can be seen in Table III. Each category is characterized by a number of paths. This values can be seen in Table III. The choice of the number of paths for each category was made based on the analysis of different OpenAPI definitions that contained a number of paths between 0 and

³⁰ISO 25010, <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/59-performance-efficiency>

³¹psutil, <https://pypi.org/project/psutil/>

151. Based on the results about the time needed to perform an analysis, an approximation was made and the ranges were defined. In the end we have, the following distribution of the files: 62% of the files belong to the very low category, 22% of files belong to the low category, 10% of files to the medium category, 4% of files to the high category, and only 2% of files belong to the very high category.

TABLE III
CATEGORIZATION OF THE FILES BASED ON THE NUMBER OF PATHS.

Type	Numbers of paths	Files
Very low	0-10	1448
Low	11-30	514
Medium	31-70	234
High	71-150	95
Very high	151+	40

VI. RESULTS

In this chapter, each research question will be addressed, and the data collected, as well as the evaluation, will be discussed.

A. RQ1: How robust is the tool regarding the successful analysis of real-world APIs?

After analyzing the 2,346 mined OpenAPI definitions, 2,300 OpenAPI definitions generated a report file. This leaves us with 46 failed OpenAPI definitions and an overall success rate of 98%. Out of these 46 OpenAPI definitions, one failed because of a faulty definition and the remaining failed due to an issue with RAM. When analyzing words, our implementation would load entries for every single word found in the source file from a dictionary, leading under circumstances to a lot of RAM being used. In our tests, it appeared as if these crashes would happen if a certain RAM threshold was reached, in our case, when going over between 5 and 7 GB RAM. In Figure 4, we can see a distribution of all violations found in the OpenAPI definitions. Most notable is that of a total of 169,061 violations, Content-Type and HTTP-401 are the most common violations, with 38,059 violations for Content-Type (22.51% of total), and 34,276 violations for HTTP-401 (20.27% of total) comprising a total of 42.78% of all violations. On the other side of the spectrum, we see barely any violations concerning verb phrases for controllers with only a total of 134 violations, for file extensions with 1,152 violations, and trailing forward slashes in the path segments of a request with 1,177 total violations. The rule for tunneling of POST and GET requests only contains 440 violations. However, this does not represent all violations found. We also need to consider our own rule in which the description should match the type of request that covers a wider range than just GET and POST and complements our rule for tunneling GET and POST requests.

In Figure 5 we can see a distribution concerning the severity of a violation. Critical and error represent violations that need to be fixed, while warnings are to be considered as recommendations. We can see 90,750 of the 169,061 total violations are critical violations which are around 53.68% followed by errors

which make up 42.10% with 71,177 violations. In total, there were only 7,134 warnings that constitute the remaining 4.22% of total violations.

In our analysis, we also tracked software quality attributes that are affected by the violations. We took the mapping of software quality attributes to rules from the Delphi study [7] conducted by Kotstein and Bogner. The distribution of these software quality attributes can be seen in Figure 6. There, we can see four software quality attributes: Usability, Compatibility, Maintainability, and Functional suitability. Maintainability makes up the largest in our figure, with 131,002 total violations falling under this quality attribute. That is a presence of 77.49% for maintainability alone. Compatibility, meanwhile, makes up 68.99% of all violations with 116,627 appearances and usability 59.72% with 100,969 appearances. Functional suitability, on the other hand, seems in only 2.37% of all violations.

B. RQ2: How effective is the tool for correctly identifying rule violations?

To shed some light on this research question, the recall and precision of the individual rules are examined. In the case of precision, a comparison is also made with the tool Zally. Unfortunately, Zally's ruleset only intersects with four of our implemented rules. Table IV gives an overview of precision (false-positives), while Table V shows the results of recall (false-negatives).

1) Rule: Forward slash separator (/) must be used to indicate a hierarchical relationship:

Precision: For this rule, 202 violations were manually examined. Of these, 192 turned out to be true-positives and only ten to be false-positives leading to a precision of 95.1% as shown in Table IV. As a path could be partially invalid by containing a part that is separated by the forward slash (/) and another that is separated by a different symbol used as a separator, such as a period (.), our implementation would throw a violation if certain characters were used. As a result, if a path would contain a file ending separated by period (.) e.g., .../my-image.jpg this rule would throw a violation although it would be a file extension violation instead. Also, the appearance of certain special characters would cause a violation, although strictly speaking their appearance alone should trigger a violation, except '?' and '#' as they mark the beginning of a fragment and a query, and should not be in the path segment of the OpenAPI definition. These two error cases describe the two types of false-positives we found in our manual examination.

Recall: Unlike for precision, we only have a recall of 44.4% for this rule. Looking at false-negatives, the results are not surprising, as the violations our implementation misses are all rules that are concerned with the hierarchy of word-segments in the path. As our current implementation of this rule does not cover any forms of word hierarchies, these cases would be missed by our implementation.

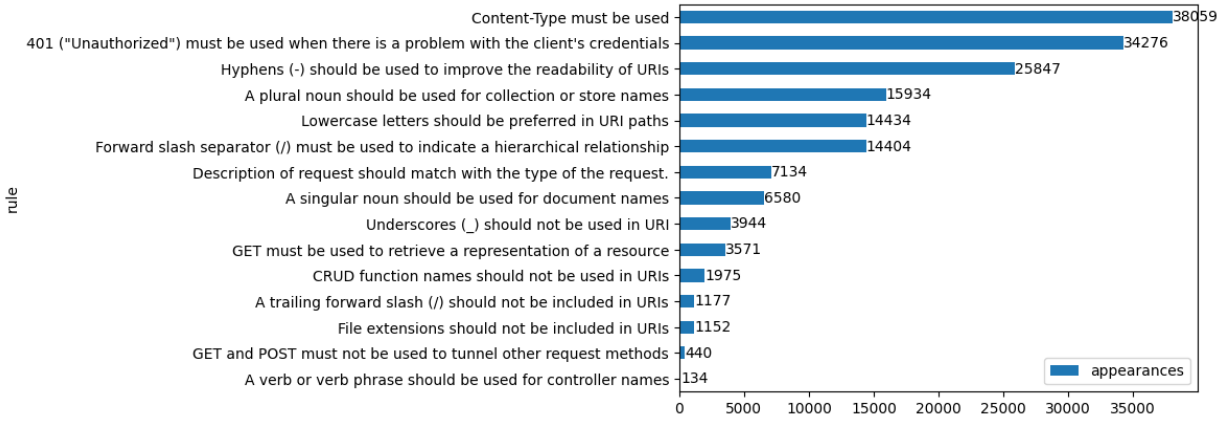


Fig. 4. Distribution of violations

TABLE IV
PRECISION RESULTS

Rule	Violations	false-positives	true-positives	Precision
Underscores (_) should not be used in URI	20	0	20	1.000
Lowercase letters should be preferred in URI paths	140	0	140	1.000
401 (Unauthorized) must be used when there is a problem with the client's credentials	358	0	358	1.000
CRUD function names should not be used in URI	21	1	20	0.952
Forward slash separator (/) must be used to indicate a hierarchical relationship	202	10	192	0.951
Content-Type must be used	219	13	206	0.941
A plural noun should be used for collection or store names	176	16	160	0.909
Hyphens (-) should be used to improve the readability of URIs	285	51	234	0.821
A singular noun should be used for document names	40	12	28	0.700
GET and POST must not be used to tunnel other request methods	86	28	58	0.674
GET must be used to retrieve a representation of a resource	9	3	6	0.667
File extensions should not be included in URIs	15	6	9	0.600
A trailing forward slash (/) should not be included in URIs	0	0	0	–
A verb or verb phrase should be used for controller names	0	0	0	–
Total	1,596	140	1,456	0.912

TABLE V
RECALL RESULTS

Rule	Violations	CLI			Zally		
		false-negatives	true-positives	Recall	false-negatives	true-positives	Recall
Trailing slash	2	0	2	1.000	0	2	1.000
GET resource	8	0	8	1.000	–	–	–
Underscores	4	0	4	1.000	–	–	–
Lowercase	6	0	6	1.000	0	6	1.000
Hyphens	9	1	8	0.889	7	2	0.222
Unauthorized	6	1	5	0.833	–	–	–
Singular noun	9	3	6	0.667	–	–	–
Content-type	6	2	4	0.667	–	–	–
File extensions	8	3	5	0.625	–	–	–
CRUD function names	13	5	8	0.615	–	–	–
Plural noun	14	7	7	0.500	0	14	1.000
Verb for controller	4	2	2	0.500	–	–	–
Forward slash	9	5	4	0.444	–	–	–
Tunnel	13	7	6	0.462	–	–	–
Total	111	36	75	0.676	7	24	0.774

2) Rule: A trailing forward slash (/) should not be included in URIs:

Precision: Since there were not as many violations found for this rule, we did not find any violations of this type in the randomly selected and manually categorized violations. However, this rule is implemented very statically; either there

is a slash at the end of the path or not. Therefore, the precision should be 100%, but there is no data available.

Recall: Since there are not so many different cases of the trailing slash, only two violations of this rule were given. Both violations were detected by the CLI, which resulted in a recall of 100%. Not unexpected, the Zally tool also detected both

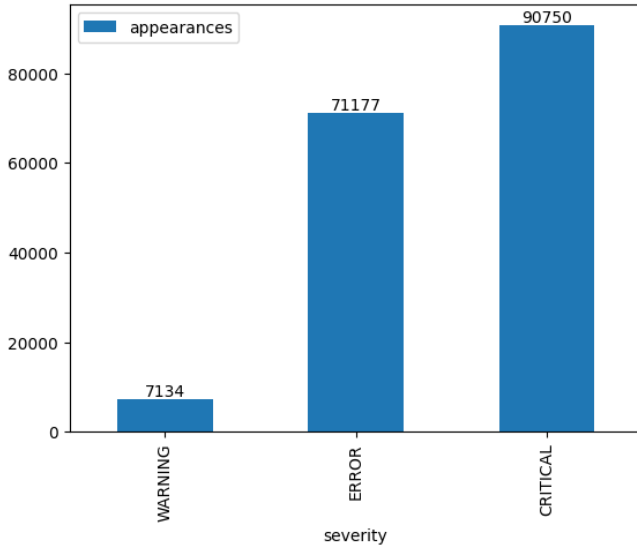


Fig. 5. Distribution of violation severity

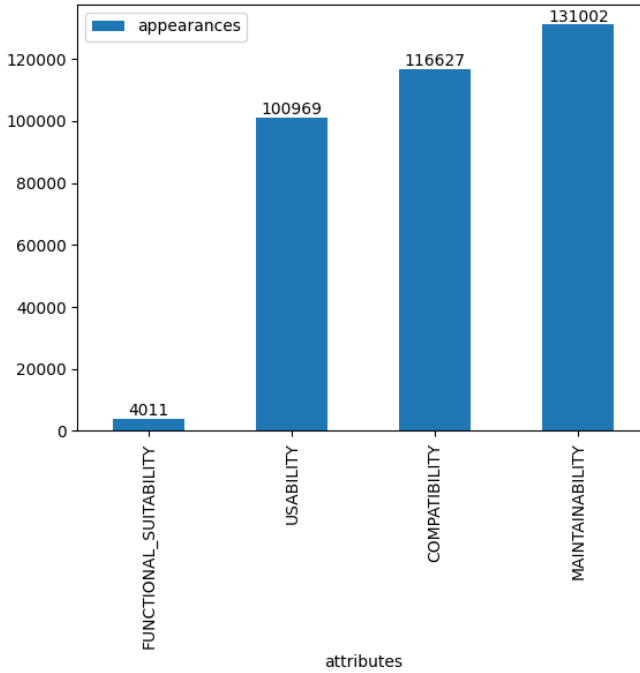


Fig. 6. Distribution of software quality attributes

violations.

3) *Rule: GET and POST must not be used to tunnel other request methods:*

Precision: For this rule, we manually examined 86 violations. In these, we discovered 28 violations being false-positives and 58 being true-positives leading to a precision of 67.4%. This result can be explained by us using a trained model to read the description of a request and compare it to its operation. It would then guess if a violation is present. As this is heavily dependent on the quality of the description and

our module making a guess, false-positives are to be expected and therefore are only displayed as a warning on the severity index.

Recall: For the recall, 13 violations were to be found of which only six were found. This outcome led to seven false-negatives and a recall of 46.2%. This result can be explained by some violations "hiding" the tunneling not in the description and using the wrong request operator but rather using a request body or query parameters to define an operator to be used with the data sent. This problem is an oversight in our implementation and would be needed to be added in future implementations.

4) *Rule: GET must be used to retrieve a representation of a resource:*

Precision: After reviewing 1596 violations by hand, we found nine violations specifically for this rule. Of these, we found three false-positives and six true-positives. This leads to a precision of 67.7%. We marked these three as false-positives as they contained a reference in the HTTP 200 response that leads to several other refs which partly contain more nested references. Our reasoning being that the references make understanding the returned object a lot harder to understand. All these references do resolve to a defined representation of a resource, though. So if we were to ignore our decision, it would lead to a precision of 100%.

Recall: For recall, we were provided with eight violations, of which all eight were detected by our implementation. This leads to a recall of 100%. This is due to the static part being easy to check. For this rule, we check if the HTTP 200 response is defined. In the case of it existing, we check if it contains a content field specifying how the returned object looks like. See also IV-H. In addition, the tool analyzed the description, which detected the incorrect use of operations to obtain a resource.

5) *Rule: File extensions should not be included in URIs:*

Precision: For this rule, a total of 15 rule violations were randomly selected for manual investigation. Nine of them are true-positives, giving a precision of 60%. However, six of them are false-positives, all of which have this structure: `.../Microsoft.Sql/servers/...`. Consequently, they have also violated the forward slash separator rule, which should be thrown exclusively. Ideas for a solution to this problem can be found in the chapter VII-B.

Recall: The experts created a total of eight violations for this rule, five of which were correctly detected by the tool. Three violations could not be detected by the CLI, giving a recall of 62.5%. One file extension (`.heic`) was not included in the list of extensions to be checked, which was later added. The other two were not recognized because the extension was not separated by a dot (e.g. `.../html`). This problem was also fixed later. Thus, the recall could be increased from 62.5% to 100%.

6) *Rule: Content-Type must be used:*

Precision: A total of 219 violations were investigated manually. 206 were true positives, while 13 were false positives. Thus, a precision of 94.1% is reached. Reasons for the false-

positives is the implementation, since if a reference to a component leads again to a reference, a violation is given. This was implemented, since otherwise the overview suffers and the individual references must be searched for.

Recall: In total, six different rule violations were provided by experts. Four of them were correctly detected by the CLI, while two of them returned a false-negative, resulting in a recall of 66.7%. It was not detected if an incorrect content type was used, since the rule only checks if a content-type was provided. This is work for the future. Also, the tool had problems when checking the content type of the parameters. This was simply missed during implementation and was found thanks to the experts. In retrospect, the error was fixed and brought the recall from 66.7% to 83.3%.

7) *Rule: CRUD function names should not be used in URI:*

Precision: A total of 21 rule violations were randomly selected by the Python script, which yielded a total of 20 true-positives through manual investigation. However, one false-positive was delivered anyway, which results in a precision of 95.2%. A false alarm was given for the word updater in the path segment, which was recognized by the CLI as a *CRUD* operation. Nevertheless, the word was added to the list containing the *CRUD* operations as substring (see [IV-H](#)), which removed the false-positive. Thus the precision was increased from 95.2% to 100%.

Recall: There were 13 rule violations, eight of which were detected by the CLI and five of which were false-negatives. However, these false-positives are due to synonyms of the *CRUD* operations: purge, fetch, retrieve, add. Since we only count direct *CRUD* operations as violations (see section [IV-H](#)), the synonyms are not detected. Nevertheless, we have added the obvious synonyms that the experts came up with to the tool, which reduces the number of false-negatives to zero. Thus, recall was increased from 61.5% to 100%.

8) *Rule: Underscores (_) should not be used in URI :*

Precision: Here, a total of 20 rule violations were examined. All violations were correctly detected by the tool, which results in a precision of 100%.

Recall: Four different violations were provided by the experts for this rule, all of which were also correctly detected by the CLI.

The reason for a recall and precision of 100% is that this rule is very static and easy to analyze. It cannot be misunderstood and must always provide a correct result.

9) *Rule: 401 (Unauthorized) must be used when there is a problem with the client's credentials:*

Precision: The team investigated 358 violations of this rule. All violations were correctly classified by the tool, and we have no false-positives, leading to a precision of 100%. This is because this rule is very static; either the 401 response is there, or it is not.

Recall: A total of six different violations were provided for these rules. Five of them were true-positives, implying a recall of 83.3%. One rule violation was not captured by the tool where the responses were used incorrectly. The 401 was indi-

cated by the description as forbidden, the 403 as unauthorized. Thus, by static examination, the 401 was present, but was used incorrectly; a way to fix this is in section [VII-B](#).

10) *Rule: Hyphens (-) should be used to improve the readability of URIs:*

Precision: Of the 285 hyphen-related violations found, only 51 were false-positives, giving a precision of 82.1% as we can see in Table [IV](#). If we look more closely at the paths that caused a false-positive, we can see that these paths contain words that are not present in our vocabulary for identifying words. The vocabulary we chose is a compromise between sufficient accuracy in correctly recognizing words and the performance aspect. We also tried using a much larger vocabulary for this rule, containing 300k more words than the one used here. The accuracy was much better, but the time required to perform an analysis was also much longer. A solution to this problem is also discussed in section [VII-B](#).

Recall: The hyphen rule also performed quite well for paths written by experts, with a recall of 88.9%. Out of nine paths, only one resulted in a false-negative, namely the path `/videogames/id`. As we explained in the precision section, this is due to the limited number of words in our vocabulary. Surprisingly, the Zally tool could only achieve a recall of 22.2%, resulting in many false-negatives, as we can see in Table [V](#).

11) *Rule: Lowercase letters should be preferred in URI paths:*

Precision: All 140 detected violations of the lowercase rule were correctly identified, resulting in a precision of 100% as we can see in Table [IV](#). This result is due to the very simple logic behind this rule, which facilitates correct identification.

Recall: The number of violations of this rule received from experts totaled six, and all were correctly identified, resulting in a recall of 100%. The explanation for this is the same as for the precision, simple logic. Interestingly, the Zally tool also had no problems identifying this type of violation, with a recall of 100% as well. This result can be seen in Table [V](#).

12) *Rule: A plural noun should be used for collection or store names:*

Precision: In the precision analysis, 176 violations were detected for this rule, of which only twelve resulted in a false-positive. This analysis resulted in a precision of 90.9% as we can see in Table [IV](#). As explained in the section [IV-H](#), implementing the logic for this rule is not straightforward and depends heavily on the model or library we use to determine whether a word is plural or singular. A possible improvement to this rule is then discussed in the section [VII-B](#).

Recall: As far as the paths received from the experts are concerned, half of the total of 14 paths led to a false-negative result. This results in a recall of 50% as we can see in Table [V](#). Three of the 14 paths contained nouns that had the same singular and plural. This case was not correctly detected by our tool. The Zally tool, on the other hand, detected all paths with zero false-negatives, resulting in a recall of 100%. However, it must also be said that many of these detections were also false-positives, leading to an ambiguous result regarding this

rule with the Zally tool.

13) *Rule: A singular noun should be used for document names:*

Precision: For the singular rule regarding precision, only 40 violations were found. Of these violations, twelve were false-positives, resulting in an accuracy of 70%, as can be seen in Table VI. The problems discussed in the plural section also apply to the singular rule, since they use the same library to determine whether a noun is plural or singular.

Recall: Only nine ways were given by experts for the singular rule. Of these nine paths, three lead to a false-negation. The reason for this is the same as for the plural rule: the three paths that led to a false-negative contain nouns that have the same singular and plural. This results in a recall of 66.7%, as can be seen in Table V.

14) *Rule: A verb or verb phrase should be used for controller names:*

Precision: Unfortunately - or fortunately - we did not find any violations of this rule in the analyzed reports, so there is no accurate data for this result, as can be seen in Table IV. This problem is not due to the random selection of reports we ran, because, as can be seen in Figure 4, no violations of this type were found in the entire analysis either.

Recall: We received only four paths for this rule from the experts, and of these four paths, two produced false-negative results. The reason for this is using verbs that are also nouns, such as "present" and "permit". These cases were not detected by the tool and no violations were found, resulting in a recall of 50%.

In summary, a total of 1,596 rule violations were manually examined for false-positives, i.e., how often the tool incorrectly returned a violation. This resulted in 140 false-positives and 1,456 true-positives, which gives a precision of 91.2%. In contrast, a total of 111 rule violations were given by the experts for the recall. With 75 true-positives and 36 false-negatives, this results in a recall of 67.6%. However, it was precisely the violations given by the experts that revealed some edge cases. A few of them were fixed directly and improved the recall; a few are for future work. Although there was only a small comparison between the rules of the CLI implementation and the tool Zally, it was interesting to see. However, when implementing the plural rule, for example, Zalando focused more on recall than on precision.

C. RQ3: How fast and resource-efficient is the tool while analyzing real-world APIs?

Table VI shows an overview of the results of the analysis of the different OpenAPI files with their respective categorization. A computer with the following specifications was used for this analysis: Ryzen 5 3600 6x 3.60GHz and 16 GB RAM. Other specifications like graphics card or memory storage were not relevant for the analysis and were therefore not considered.

Regarding the time needed to perform an analysis, we can see that for a *Very low* type we need only seven seconds for an analysis. This is also related to the number of paths a file has, here we have a median of three paths per file. As the

number of paths increases, the time required for an analysis also increases. As we can see, for the *Low* type we need three times as much time, 23 seconds, for an analysis compared to the *very low* files, but we also have almost six times as many paths, 17, as the previous category. For the *Medium* category, we need almost 2.5 times, compared to the *Low* types, and eight times, compared to the *Very low* types, as much time, 54 seconds, for an analysis as for the previous types. The number of paths here is 43 compared to the 17 of the *Low* and 3 of the *Very low*. With 2.5 times the number of paths, the time needed for an analysis also increases by 2.5 times compared to the previous category. Continuing with the *High* category, we can see a similar pattern to the previous one. An increase in the number of paths corresponds to an increase in the total time required to complete an analysis. Here, 117 seconds are required, with the median number of paths being 89.5. Compared to the previous category, the *Medium*, the time required has increased by 2.2 times and the number of paths by 2.1 times. With the last category, *Very high*, the same pattern can be seen as with the previous categories. The increase in the number of paths corresponds to the time needed to finish an analysis. Here we have 2.3 times the number of paths (206 compared to 89) in relation to the *High* category, which corresponds to a 2.5 times increase (301 seconds compared to 118) in the time needed for the analysis.

There are also some differences in memory utilization (RAM). The largest difference is between the *Very low* type and the others: here we have a median of 497 MB compared to 957, 1,067, 1,111 and 1,176 MB for the other categories. While for the other categories we still have a difference, but not as big as the previous one, "only" 200 MB more memory usage between the *Low* and *Very high* types. CPU utilization is not so relevant to the analysis, although with some peaks the median is the same for all categories, 8.4%.

File size also increases from one category to the next, but is not as relevant as the number of paths. This is because we can have files with a size of 1.6 MB in the *Very low* category, but the time required to perform an analysis is not comparable to that in the *Very high* category.

In summary, there is a recurring pattern between the time needed to perform an analysis and the number of paths in an OpenAPI definition. We also saw that the majority of files, Table III, belonged to the *Very low* category, which has a very good performance efficiency, with a median of seven seconds to perform an analysis and relatively low memory usage, 496 MB. This means that our tool for this category can perform a relatively fast analysis. Nevertheless, the analysis of a file of the other category is also efficient. The time span between 23 seconds (*Low*) and two minutes (*High*) is still acceptable to wait for the completion of an analysis. The six minutes needed for an analysis of the files belonging to the *Very high* category, on the other hand, could be barely acceptable for a user.

TABLE VI
PERFORMANCE RESULTS

Type	Time Median	Size Median	Paths Median	Memory Median	CPU Usage Median
Very low	7.25 s	0.03 MB	3	497 MB	8.4%
Low	23.52 s	0.14 MB	17	957 MB	8.4%
Medium	54.73 s	0.36 MB	43	1067 MB	8.4%
High	117.77 s	0.69 MB	89,5	1111 MB	8.4%
Very high	300.96 s	1.6 MB	206,5	1176 MB	8.4%

VII. DISCUSSION

Now, that we have looked at all the results found in our study, we would like to discuss some of these in detail. We then address possible future implementations of the CLI, and finally, describe the threats to validity of the study.

A. Results

In the following, each research question is addressed and the results obtained from the study are discussed.

1) *RQ1: How robust is the tool regarding the successful analysis of real-world APIs?*: In section V-B1 we defined robustness as, how well our tool performs in processing OpenAPI definitions without producing an error. From section VI-A we gathered an overall 98% success rate of analyzing OpenAPI definitions and creating a valid report file. Of 2,346 OpenAPI definitions, 2,300 were analysed correctly, while only 46 caused an error. Therefore, with the 98% success rate, we claim that the tool is quite robust, with the current set of rules implemented. But it is clear, there are still things that could be improved upon to make this tool even more robust. Out of the 46 OpenAPI definitions, 44 were due to a RAM threshold being reached or due to bugs in our application. Many of those crashes that were due to RAM, were on files that were large and had a lot of path end points. Also, the rules in which the crash of our tool occurred were rules doing text analysis. In particular the rules for *Forward slash*, *Plural noun*, *Singular noun*, and *verb for controller* caused crashes due to RAM issues. As already mentioned in V-B1, the computer used to perform the analysis had up to 7 GB RAM usage for some OpenAPI definition. We therefore suspect these files containing a significant amount of different words, that need to be labeled by the model as a plural, singular or verb, which then results in a lot of RAM being occupied. Also the space, occupied in the RAM, not being cleared after each rule could also contribute to the RAM limit being reached. A way we might be able to circumvent that problem would be by setting a custom internal RAM threshold for the tool. If this threshold is reached, a routine could check for data in the RAM which hasn't been used for a certain time frame and remove it from the RAM to free memory.

2) *RQ2: How effective is the tool for correctly identifying rule violations?*: For this research question, the recall and precision of the tool were investigated to get a statement for its effectiveness. Overall, the tool has a precision of 91.2%, which should be sufficiently good. This examination of 1,596 rule violations was the first large-scale manual evaluation we have done with the tool. As already described in the chapter on

results, we came up with some improvements and problems, how the precision could be further improved. Some rules were improved directly, like the *CRUD function name* rule, but some are meant for the future work, like implementing a hierarchy between the rules (see section VI).

However, a few rules cannot be greatly improved by the logic implemented so far. Three rules (*singular/plural noun*, *verb for controller*, *tunneling*) depend on probabilities from a model, which cannot guarantee a precision of 100%.

The *content-type* rule has a static implementation, but is not 100% precise. This is because the implementation doesn't allow nested references to components. Therefore, the *content-type* is defined in the components, but nested, which leads to a false-positive.

Nonetheless, there is potential for improvement with the *hyphens* rule, which only works as well and precisely as its dictionary. Nevertheless, performance would suffer with a larger dictionary, which is why a smaller one was deliberately chosen.

Regarding recall, the tool did not perform as well with 67.6%. One reason for this could be that rule violations for this metric were created by experts and often contained edge cases that were difficult for the tool to identify. We assume that most of these edge cases should be rare in industry environments, which may have made it difficult to detect them in advance of the study. However, based on the provided edge cases, we were able to further improve several rules. Some rules (*content-type*, *file extension*, *CRUD function names*) were already adjusted directly after the analysis, which raised the recall to 75%. No new logic was added; only the existing solution was adapted.

Some rules were also designed to not detect all false-negatives, since this is usually simply not possible. Especially the *tunneling* rule is extremely difficult to examine statically. Because of the importance of the rule, however, a static implementation was selected, which could deliver some violations, but not all. Particularly with the *hyphens* rule, the implementation is only as good as the dictionary, which in turn limits the detection of false-negatives. For the *singular* and *plural noun* rule, as for the *verb* rule, the model to determine the type of word is not 100% precise. Nevertheless, they improve the model continuously, as can be seen on the web page³². It is also difficult and complex to reduce the number of false-negatives for the *forward slash* rule. Only for the implementation of the necessary hierarchy of the segments there are whole studies for this, see next section VII-B.

³²OpenNLP Models, <https://opennlp.apache.org/models.html>

The *file extension* and *CRUD function name* rules, need to be adjusted by usage first, so that they can detect more false-negatives. Both are based on a list of words for which the path is searched. Neither list is complete and only provides a base that can and should be extended.

The comparison of the four rules (*lowercase*, *trailing slash*, *hyphens*, *plural noun*) between the CLI and the tool Zally was interesting. Not surprisingly, there was a recall of 100% for the *lowercase* and the *trailing slash* rule, as these are statically very easy to check. Most surprising, however, was the result of the *hyphens* rule and the *plural noun* rule. For these rules, Zalando seems to have taken two different approaches. The *hyphens* rule was designed to have as few false-positives as possible, which thus, suffers from recall. In contrast, with the *plural noun* rule, there were no false-negatives, but a lot of false-positives. Thus, for both of these rules, there was a wide margin between recall and precision.

Overall, it can be concluded that the values of recall and precision should be brought closer together to ensure a good effectiveness of the tool. Currently, the tool tends to be more conservative, showing few false-positives and therefore, risking a higher number of false-negatives. For recall, it would be interesting to have the number of false-negatives for the same data set as for the precision. However, this is very difficult or even impossible, because false-negatives cannot be counted easily.

3) *RQ3: How fast and resource-efficient is the tool while analyzing real-world APIs?:* As we saw in the section VI-C, the analysis shows good performance for the *Very low* and *Low* categories. With a median of 7 seconds for *Very low* and 23 seconds for *Low*, we can conclude, that the tool performs quite well from a timing perspective. These two times are overall a good indicator for a more general performance, since 84% of the total analyzed files belong to these two categories. For the remaining categories, the time needed to perform an analysis became much higher, arriving at 300 seconds required for the *Very high* category. This could lead to dissatisfaction from the user's point of view. But on the other hand, the files analyzed in these categories are much larger than the first two categories, and it is also much rarer to analyze this type of OpenAPI definition, which contains a number of paths above 31. Overall, the results are quite good regarding the time aspect.

As for resource usage, we also see a good performance for files belonging mainly to the first category, *Very low*, where the median is 497 MB of RAM used per analysis. Since the majority of the analyzed files belong to this category, this result is quite good and indicates a low resource usage for most of the files that this tool will analyze. For the remaining categories we have a similar RAM usage, from 957 MB to 1,176 MB for the *Very high* category. This amount is also not that high and can work very well on most computers³³, but might run into problems on computers with a total amount of RAM of 4 GB. For the CPU usage, a similar value results for all categories,

namely 8.4% of the total CPU usage. This result indicates that the CPU is not a bottleneck in performing the analysis with this tool, and that RAM usage plays a more important role.

In summary, the tool performed particularly well for the *Very low* category, which also represents the majority of the files analyzed. As for the time aspect, the analysis of files of the *Low* category is also quite performant.

B. Future Implementation

Although, the implementation of Massé's design rules [3] has shown some promising results. However, the study performed on robustness, effectiveness and performance has shown there is much to improve upon. For one, there seem to be violation cases, that match multiple rules. For example, a violation of the *underscore* rule can also be a violation of the *hyphens* rule, if underscores are used to separate words to make the path segment more readable. At the same time, it also could be a *forward slash* violation if the underscore is used as an intended separator. It becomes clear, that some sort of hierarchy within the rule set could be useful to make the report for a user more understandable. So, instead of three violations for the same violations subject, the implementation would only list the most accurate violation of the three. This would also fix an unwanted interaction we encountered between the *forward slash* rule and the *file extension* rule, which we'd consider a false-positive. If a path contained a file extension at the end of a path, a *forward slash* violation would erroneously be thrown.

We also believe it would be beneficial to add a description analysis for many rules. For example, in the case of the *unauthorized* rule, some violations from the experts would mix up HTTP 401 and HTTP 403. Thus, they use the HTTP 403 responses for *unauthorized* and HTTP 401 for *forbidden*, instead as intended HTTP 401 for *unauthorized* and HTTP 403 for *forbidden*. Here, we could read the description field of those responses and check if the response might not be used as intended. However, a large data set must be collected for training a model, which might not be that easy.

As we have already mentioned in the section IV for rule implementation, the *forward slash* rule currently does not implement any form of hierarchy analysis. As a result, in the effectiveness analysis, all violations related to hierarchy were not found. A possible way of implementing a hierarchy check could probably be derived from recent works by Palma et al. in 2015 [6] and in 2017 [4]. They have done multiple in depth analysis of the rules by Massé, especially in the lexical context. They also created their own implementations of these rules, and could therefore be of great guiding stone for an implementation of hierarchy checking.

After running the OpenAPI analysis over the definitions of the experts, it became clear that the *content-type* rule would also benefit from some improvements. When it comes to *content-type* definitions, always the most descriptive type should be used. For example, if the object returned from a request is represented in JSON format, the *content-type* field in the OpenAPI definition should also specify JSON as its *content-type*. Currently, our implementation of this

³³Steam survey, <https://store.steampowered.com/hwsurvey>

only analyze if a `content-type` definition is present and does not check if the `content-type` defined is the most accurate descriptor of the data representation of a response. It would also be worth considering to have a deeper analysis of nested references in the components. Presently, we throw a violation if a reference points to another reference for object definitions in responses. The problem with our current approach is, complex objects could be treated as a violation of this rule, even though their `content-type` definitions are correctly defined.

The current implementation would also benefit greatly from more dynamic analysis of rule violations. Currently, we only have dynamic analysis for one rule, namely the *unauthorized* rule. But several of the implemented rules have use cases, that could be covered with a dynamic analysis, but were not implemented due to time constraints. For example, the rule *"GET should retrieve a representation of a resource"* could be dynamically checked, by sending a GET request and checking if the returned message contains the in the OpenAPI definition specified object. Although, we have not implemented many dynamic checks for the rules, we have set the ground work for future implementations of these dynamic rule checks, like checking the `content-type`. More information on the implementation of dynamic analysis can be found in [IV-E](#).

The *hyphens*, *singular/plural noun*, and *tunneling* rules were the rules that were a bottleneck for the tool regarding the performance aspect. These rules required a lot of resources to perform the analysis of a path. This problem is due to the implementation and use of a model in the case of the *singular/plural noun* and *tunneling* rule or a dictionary in the case of the *hyphens* rule. One solution to this problem would be to implement a cache in which analyzed path segments or paths are stored. When a new path segment or path exists in the cache, there is no need to use the model or look it up in the dictionary to get the output required by these rules. A read operation from the cache would be much faster than doing all the processing of the given input again. This solution could lead to an overall positive improvement in the tool's performance.

By asking experts about rule violations, edge cases brought out weaknesses in the implementation. These have already been described in the results chapter. If these were fixed in the future, further questioning of the experts would be helpful to find further weak points. Additionally, it would be interesting to see how well the recall has changed through a new iteration of improvement work.

C. Threats to Validity

To collect valid data, a good sample is needed that provides a broad representation of the public. When selecting samples, as well as when collecting data from them, threats to validity can arise, which are described in the following.

1) *Insufficient or non-representative sample*: When browsing through the OpenAPI repository *"apis.guru"*, it quickly becomes apparent that many of the definitions provided are

from Amazon. Since the sample should be representative of the global public, the definitions should also come from different companies, and not just from a few large ones, like Amazon, Google, and Microsoft. These would all share a lot of similarities that could falsify the overall OpenAPI definition landscape. After further investigations, we conclude that even though several OpenAPIs were publicized by larger companies, most of the OpenAPIs we mined were still from different sources. In total, from the 2,346 mined OpenAPI definitions only 7.5% were from the larger companies mentioned above (Amazon 5.7%, Google 1.5%, Microsoft 0.3%). This leads us to believe that this set of OpenAPI definitions is a suitable sample to represent the OpenAPI definition landscape.

The sample of OpenAPI documents consists only of publicly available ones. Therefore, care must be taken when generalizing the results to private definitions. Moreover, as we have limited ourselves to only consider OpenAPI definitions of RESTful APIs, we can not make any statement about any API definition not following the OpenAPI specification.

2) *Data collection*: As we wrote the script for the analysis of the OpenAPI definitions, we tried to make sure that there are no other external factors that could affect the results. However, a system is never error-free, and some errors may inevitably occur, especially if the complete analysis of the OpenAPI definitions runs for two days. For example, in the case of performance analysis, it could be that the Python environment in which the script was executed required more RAM and CPU, which automatically lead to higher resource usage by the tool. Through the tool used to measure the performance of the analysis, it was possible to isolate the Java process responsible for running the tool and thus obtain an accurate result.

In case of the robustness analysis, using URLs instead of local paths to files runs the risk of files no longer being available. For instance, the file could have been moved and the URL could be no longer valid. Also, loss of internet connection can cause the tool to fail as the URL cannot be reached in the time of no internet connection. To avoid these complications, we stored all OpenAPI definitions locally. This way, if a definition was no longer available online, we could still analyze the locally stored definition.

As sending requests to an API often requires some form of authentication, we do not perform any dynamic analysis of these OpenAPI definitions in the analysis. The reason for this is, that there is no uniform handling of the authentication of the requester. Some would offer a path to which a requester could send a request to receive an authentication token but even these could look very different between APIs. Others would require to provide login data to receive any authentication tokens. These would either be provided via HTTP headings or require more human interaction by opening a login prompt on the user's screen. Finally, most OpenAPI specifications failed to provide any information on how to authenticate. Often, there would be no security definition, but on making a request, the server would deny the request. Therefore, we cannot make any statement about the performance of the implementation con-

cerning dynamic analysis since it is not feasible to gather all authentication schemes of all 2,346 OpenAPI definitions. This also means, parts of the implementation of the "unauthorized" rule are currently not well tested.

For the precision analysis, we have tried to gather a representable sample size of violations, by randomly selecting violations and manually checking these violations for correctness. Although we have done this, it is still possible that we might have missed certain false-positives. Therefore, the calculated precision for the rules could potentially deviate from the actual value, if we were to check the entire list of violations. However, a complete manual investigation of 131,002 violations is not feasible, which is why we examined a fairly large data set of 1,596 violations (1.2%) instead. For this, we randomly selected 90 OpenAPI definition reports, from which we checked up to 30 violations each. These 30 violations were also randomly selected and then manually analyzed for false-positives. Thus, a statement about the complete data set should be possible through the investigated violations.

Also, we need to add that all data provided by the experts for the recall analysis were fabricated violations. This means that all OpenAPI definitions we received from the experts were not used in an industrial environment, but rather were explicitly created to test the rule implementations. Therefore, there is always the chance that these might not be representative enough of an industrial environment because they show only the edge cases. Nonetheless, catching those edge cases is important to increase the overall quality of OpenAPI definitions and are likely to appear in some form or another.

Another problem with the generated rule violations from the experts was that some of them were incomplete. For example, this was mostly the case regarding the description and summary of a path. However, these then had to be provided by us as researchers. Especially with the *tunneling* rule, this was not optimal. Nevertheless, just by the path, the parameters, and the responses, we could derive the description quite well. Therefore, the results should not be influenced by the addition of the incomplete OpenAPI definitions.

Due to time constraints and because there were not so many different systems available on which we could perform the analysis, we only performed the analysis of robustness and performance with one specific system configuration. As for the performance aspect, this could lead to better or worse results depending on the configuration of the systems. A faster CPU and RAM could give better results compared to an old system that needs more time to process and perform the analysis. The systems with which the analysis was performed are part of the current average standard that most people have. This would mean that similar results can also be obtained with other systems. A Steam survey³⁴ of 120 million users revealed the specification of a computer that most people have. This result led us to believe that other users of the tool should not have much trouble performing an analysis and could expect results similar to those in Table VI. As for the robustness aspect,

depending on the RAM and the size of the OpenAPI definition file, more errors could possibly occur, as reaching the system-defined RAM threshold will cause the tool to crash.

In the analysis of the *tunneling* rule, where the description is used as the most important point in deciding whether a path represents a violation or not, the model used to analyze this data was also trained on data with paths from the same sources used for the analysis. This produces an immediate true-positive result when the same path is analyzed with the same description that was used for training. The training data directly from the same source used for the analysis amounts to about 2,400 descriptions. These descriptions represent 1% of the total violations found by the tool, and an even smaller fraction of the total paths analyzed with the corresponding descriptions. This should also not be a major external factor that could distort the results obtained.

VIII. CONCLUSION

RESTful APIs and web services have become very popular in the industry and represent one of the commonly used ways to expose functionality. Despite their popularity there is no standardized set of design rules, causing the API landscape to be very heterogeneous. In this work, we presented a tool-based approach to analyze and check OpenAPI definitions of RESTful APIs with the goal of providing developers with a tool to write cleaner and better understandable OpenAPI definitions. For this, we implemented 14 API design rules defined by Massé and tested the implementation on 2,346 different publicly available OpenAPI definitions and 14 fabricated OpenAPI definitions by industry experts. We then used the results from this analysis to give a statement about three quality indicators, namely the robustness of the tool to parsing potentially erroneous real-world APIs, the effectiveness of the tool in finding rule violations, and lastly performance efficiency of the tool during analysis.

The collected data suggests that the tool is quite robust to errors, with a success rate for reading and analyzing OpenAPI definitions of 98%. We did encounter issues with RAM usage, as the rule implementations for text analysis would require a lot of space, causing errors if RAM threshold would be reached. For the effectiveness of the tool, a slightly larger margin lies between precision and recall. For precision, a total of 1,596 rule violations from the robustness analysis were manually validated. Thereby, 140 false-positives were detected, which leads to a precision of 91.2%. For the recall, on the other hand, seven experts were asked about rule violations. 111 violations were collected for all rules, with a total of 36 false-negatives, resulting in a recall of 67.6%. However, one reason for the margin could be that the violations from the recall analysis were created by experts and were often edge cases, which are particularly difficult for the tool to detect. On the other hand, the value from the precision analysis reflects the violations from the industry environment. Still, the results indicate a positive effectiveness of the tool on the analyzed OpenAPI definitions. As for the performance aspect, the data from the results suggest that our tool would

³⁴Steam survey, <https://store.steampowered.com/hwsurvey>

perform quite well for most OpenAPI definitions on most of today's computers. The time required to perform an analysis for most definitions ranges from 7 to 23 seconds, and RAM requirements range from 497 MB to 957 MB.

As our implemented tool represents a prototype and has only limited rule implementation, there are still many additions and improvements that can be done. Through the evaluation, some ideas for improvement have emerged, which could be added in the future. Additionally, many of the implemented rules do not perform any dynamic analysis, which makes them miss certain violation cases. While some foundations for implementing those dynamic checks are done, we did not manage to implement them completely. Therefore, future work could focus on adding new functionality on top of this work. Out of the 28 rules classified as important rules by Kotstein and Bogner [7], we only were able to implement 14 rules, leaving another 14 rules that could be implemented in this tool. Furthermore, after improving the tool, another study can be performed with experts, to increase the recall and precision even further. Finally, future research could investigate the quality of OpenAPI documents and how they behave with the number of paths in a definition.

REFERENCES

- [1] D. Jacobson, G. Brail, and D. Woods, "Apis: A strategy guide," 2011.
- [2] C. Pautasso and E. Wilde, "Restful web services: principles, patterns, emerging technologies," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 1359–1360.
- [3] M. Mark, *Rest api design rulebook*. O'Reilly, 2012.
- [4] F. Palma, J. Gonzalez-Huerta, M. Founi, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Semantic analysis of restful apis for the detection of linguistic patterns and antipatterns," *International Journal of Cooperative Information Systems*, vol. 26, no. 02, p. 1742001, 2017.
- [5] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of rest patterns and antipatterns: a heuristics-based approach," in *International Conference on Service-Oriented Computing*. Springer, 2014, pp. 230–244.
- [6] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, and G. Tremblay, "Are restful apis well-designed? detection of their linguistic (anti) patterns," in *International Conference on Service-Oriented Computing*. Springer, 2015, pp. 171–187.
- [7] S. Kotstein and J. Bogner, "Which restful api design rules are important and how do they improve software quality? a delphi study with industry experts," in *Symposium and Summer School on Service-Oriented Computing*. Springer, 2021, pp. 154–173.
- [8] T. Pfaff, "Haben design-regeln einfluss auf die verständlichkeit von restful apis? ein kontrolliertes experiment," 2021.
- [9] N. M. Devadiga, "Tailoring architecture centric design method with rapid prototyping," in *2017 2nd International Conference on Communication and Electronics Systems (ICCES)*. IEEE, 2017, pp. 924–930.
- [10] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000, aAI9980887.
- [11] J. Ponelat and L. Rosenstock, *Designing APIs with Swagger and OpenAPI*. Manning, 2022. [Online]. Available: <https://books.google.de/books?id=BSp0zgEACAAJ>
- [12] M. Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*. Manning, 2021. [Online]. Available: https://books.google.de/books?id=A92_zQEACAAJ
- [13] F. Petrillo, P. Merle, F. Palma, N. Moha, and Y.-G. Guéhéneuc, *A Lexical and Semantical Analysis on REST Cloud Computing APIs: 7th International Conference, CLOSER 2017, Porto, Portugal, April 24–26, 2017, Revised Selected Papers*, 07 2018, pp. 308–332.
- [14] H. Brabra, A. Mtibaa, F. Petrillo, P. Merle, L. Sliman, N. Moha, W. Gaaloul, Y.-G. Guéhéneuc, B. Benatallah, and F. Gargouri, "On semantic detection of cloud api (anti)patterns," *Information and Software Technology*, vol. 107, pp. 65–82, 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S095058491830226X>
- [15] J. Bogner, S. Wagner, and A. Zimmermann, "Collecting service-based maintainability metrics from restful api descriptions: Static analysis and threshold derivation," 09 2020.
- [16] K. Chandrasekar, "Workplace environment and its impact on organisational performance in public sector organisations," *International journal of enterprise computing and business systems*, vol. 1, no. 1, pp. 1–19, 2011.
- [17] M. Klopotek, "The advantages and disadvantages of remote working from the perspective of young employees," *Organizacija i Zarzadzanie: kwartalnik naukowy*, 2017.
- [18] J. E. Hannay, T. Dybå, E. Arisholm, and D. I. Sjøberg, "The effectiveness of pair programming: A meta-analysis," *Information and software technology*, vol. 51, no. 7, pp. 1110–1122, 2009.
- [19] A. McCallum and K. Nigam, "A comparison of event models for naive bayes text classification," in *AAAI Conference on Artificial Intelligence*, 1998.
- [20] R. P. Jetley, P. L. Jones, and P. Anderson, "Static analysis of medical device software using codesonar," in *Proceedings of the 2008 workshop on Static analysis*, 2008, pp. 22–29.
- [21] J. M. S. França and M. S. Soares, "Soaqm: Quality model for soa applications based on iso 25010," in *Proceedings of the 17th International Conference on Enterprise Information Systems - Volume 2*, ser. ICEIS 2015. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, 2015, p. 60–70. [Online]. Available: <https://doi.org/10.5220/0005369100600070>
- [22] W. P. Alexander, R. F. Berry, F. E. Levine, and R. J. Urquhart, "A unifying approach to performance analysis in the java environment," *IBM Systems Journal*, vol. 39, no. 1, pp. 118–134, 2000.