Web APIs Structures and Data Models Analysis

Souhaila Serbout
Software Institute
Università della Svizzera Italiana
Lugano, Switzerland
souhaila.serbout@usi.ch

Fabio Di Lauro
Software Institute
Università della Svizzera Italiana
Lugano, Switzerland
fabio.di.lauro@usi.ch

Cesare Pautasso
Software Institute
Università della Svizzera Italiana
Lugano, Switzerland
c.pautasso@ieee.org

***Abstract*—Microservice architectures emphasize keeping components small, to foster autonomy, low coupling, and independent evolution. In this large-scale empirical study, we measure the size of Web API specifications mined from open source repositories. These APIs are modeled using the OpenAPI Specification (OAS), which, in addition to documenting the offered operations, also contain schemas definitions for the data exchanged with the API request and response message payloads. This study has as a goal to build empirical knowledge about: (1) How big and diverse are real-world web APIs both in terms of their operations and data, (2) How different API structures use and reuse schema definitions. By mining public software repositories on Github, we gathered 42,194 valid OAS specifications published between 2014-2021. After measuring the size of API structures and their data model schemas, we found that most APIs are rather small. Also there is a medium correlation between the size of the APIs' functional structures and their data models. API developers do reuse schema definitions within the same API model.**

***Index Terms*—Web API, OpenAPI, Data Model Schemas, API Size Metrics, API Structure**

I. Introduction

Every open software architecture has an API (Application Programming Interface) which makes it possible for external clients to access the services it provides. Unless self-contained, every software architecture also reuses and depends on multiple external APIs. APIs are found in microservice architectures [14], making it possible to clearly delimit the boundary of each service [14], which can be independently operated and autonomously evolved. In this paper we focus on Web APIs, which make use of the HTTP protocol to exchange messages across the network [12]. These are commonly found in all kinds of application domains, for example, to remotely access and provision Cloud computing infrastructure, read and post media on social networks, or enable same-day payments in digital banking [9].

APIs are a critical design element which affects how developers select, use and reuse a software component [15]. Writing accurate and detailed API documentation is essential in order to bridge the gap between API providers and their consumers and sharing the needed knowledge to correctly use them. The available documentation and the perceived ease of use are also a relevant factor for developers to decide to use or not a specific API [8] as well as other aspects such as the quality of service guarantees [22], popularity, provider reputation, performance [1], pricing plans [4] and security.

This work is funded by the SNSF, with the API-ACE project nr. 184692.

A growing number of Web APIs are described using the OpenAPI language [18]. Previously known as Swagger, this JSON/YAML-based Interface Description Languages (IDL) is used to enumerate the operations offered by the API (in terms of Web addresses or URI path templates and HTTP method combinations) as well as to describe the data model of the API (mainly in terms of JSON schemas). OpenAPI was originally introduced to automate documentation generation tasks. Nowadays it is at the core of a growing API tooling ecosystem, including support for code generation [7], testing [6], as well as model analysis [2, 17] and validation.

In this paper we present an empirical study [3] over a large collection of Web APIs performed by statically analyzing their OpenAPI descriptions mined from open source repositories. We are interested to answer the following research questions:

Q1: How big are the API models?
Q2: Does the API style affect the API size?
Q3: What is the relationship between the sizes of the functional structure and data structure of an API?
Q4: How developers reuse schemas within the same API?

Observing the size and the style of real-world Web APIs is important as they are among the factors impacting API governance [5], developer experience [8], learnability [15] as well as service granularity [19].

To answer these questions we first need to define how to measure the size of a Web API specification. We then proceed to characterize the APIs in the collection according to the popularity of the artifacts describing them, as well as to how the APIs make use of the HTTP protocol [16] (distinguishing between different API styles: read-only or read-write Open Data, RPC, CRUD and REST).

We find that not all API specifications include both data and operations. Also, over the entire collection, excluding the empty API models, the size of the API structure and the API data model have a correlation which ranges from 0.58 to 0.63, depending on the metric used. Finally, developers are making use of schema reuse constructs, with some APIs reusing schema definitions across different operations up to 8 times.

The rest of this paper is structured as follows. Section II presents an overview of the collected API artifacts and how they were obtained by mining public GitHub repositories.

Section III defines how to measure the size of API description artifacts. Section IV describes the analysis results, which are later discussed in Section V. We list threats to validity in section VI. Section VII summarizes the related work, before we conclude in Section VIII.

II. Dataset

Before presenting the analysis results, in this section we describe how we obtained and prepared the dataset by crawling public code repositories on GitHub which included one or more OpenAPI description files (OAS).

A. Dataset preparation pipeline

The automatic process of creating our dataset and filtering shown in Figure 1, involves the following steps:

1) Mining GitHub we collected 109,587 OAS files from 66,971 different public software repositories.

2) In version control platforms such as GitHub, users are allowed to fork repositories, which caused redundancy in the collected dataset. In order to avoid this potential bias in our metrics-based analysis, we remove duplicate artifacts reducing the total amount of OAS to 56,411 files.

3) The OAS can be described in a single file or organized into different files using references. In case these references are not available or the file does not pass other validation rules according to the specification [18], we discard it. This leaves 42,194 OAS descriptions that conform to the OpenAPI specification, according to standard validation tools.

4) We extracted API structure and data models and their corresponding data types from the valid API descriptions, obtaining a collection of 259,869 different types, 150,027 resource paths, and 194,812 operations.

5) We further filtered the dataset to keep 31,118 valid, unique, non-empty APIs which make use of JSON payloads.

6) Finally, for each API description, we computed the metrics described in Section III and performed some basic statistics over them to answer the research questions.

B. Dataset description

The oldest API descriptions in the collection are from November, 27th 2014. Figure 2 shows how the size of the API collection has been increasing over time since then. Most of the API descriptions in our dataset (90.5%) have been updated at least once since 2019, 81.3% have been updated at least once since 2020, and 55.7% have been updated in 2021. This

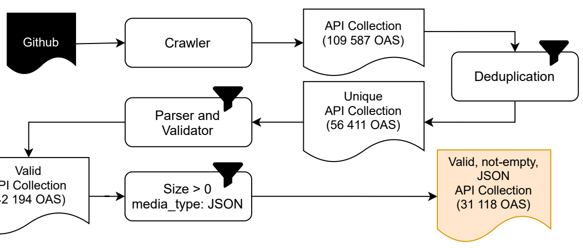

Github
Crawler
API Collection (109 587 OAS)
Deduplication
Unique API Collection (56 411 OAS)
Parser and Validator
Valid API Collection (42 194 OAS)
Size > 0 media_type: JSON
Valid, not-empty, JSON API Collection (31 118 OAS)

Fig. 1: Data preparation pipeline

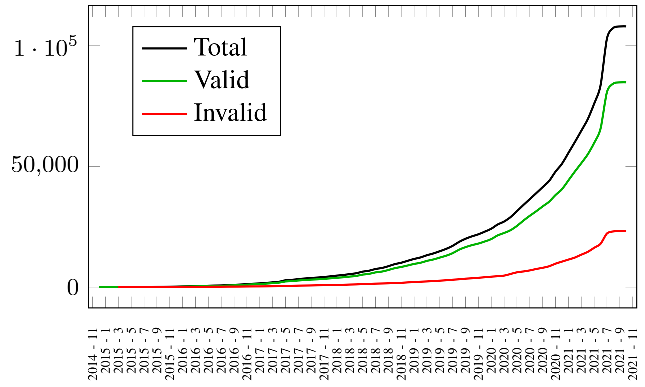


Fig. 2: Monthly cumulative distributions of valid and invalid OAS files based on the date of their most recent version

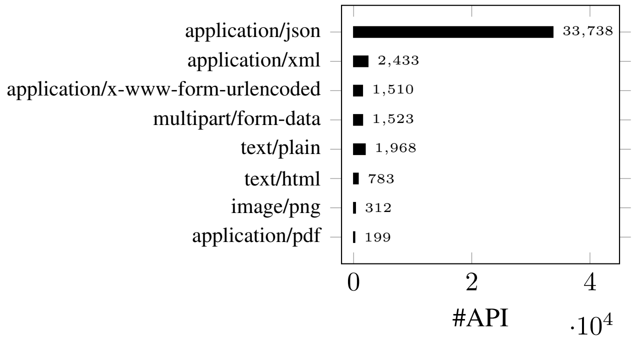


Fig. 3: Number of APIs with valid specification where a specific media type appears

shows the growing adoption of OpenAPI as an IDL to describe Web APIs.

Overall, the dataset size of the filtered collection of valid specifications is $1.2GB$ with an average OAS file size of $24.3KB$. The total number of distinct providers in the collection is 331. The dataset was collected from 01.12.2020 until 24.10.2021.

In terms of data models, only a minority of APIs does not use message payloads represented in the JSON format (Figure 3). In this study we focus on the majority of APIs in our collection, the ones with a JSON Schema data model.

III. Metrics

A. Metrics definition

In this analytics study we define a set of metrics that can be applied to measure the size of different parts of an OAS specification. We distinguish between basic metrics and indicators. Basic metrics are directly computed by counting the elements found in each OAS description while indicators are derived from the metrics to further characterize each API.

The main goal of the metrics are to measure the size of a Web API described using OAS, in particular concerning its structure and data model schema. Additionally, we are interested to measure the rate of internal reuse, e.g., how many data model schema types are shared among different operations, as well as to use the number of clones as a proxy to estimate the popularity of an API.

- **API Structure Size Metrics**
  - *#Paths*: Total number of resources published by the API
  - *#Operations*: The total number of HTTP methods for all resources provided by the API.
  - *Depth*: Maximum path length (in terms of path segments)
  - *Breadth*: Number of top-level distinct path segments
  - *Operations per Path* (OPP): $\frac{\#Operations}{\#Paths}$.
  - *HTTP Methods*: The number of distinct HTTP methods across all operations provided by the API.
- **API Data Model Size**
  - *#Defined Schemas* ($\#\mathcal{DS}$): schemas can be embedded in the operations descriptions (e.g, request body of a PUT or POST method, any method response) or listed separately in the `components` section in the case of OAS 3 or in the `definitions` section in the case of OAS 2. The *#Defined Schemas* metrics counts the total of all the schemas defined in the OAS file, either embedded in an operation description or separately defined.
  - *Embedded Schemas* ($\mathcal{ES}$): the proportion of embedded schemas relative to the total number of defined schemas.
  - *#Used Schemas* ($\#\mathcal{US}$): the number of the defined schemas that are used at least once, i.e., they are referenced directly or indirectly from an operation request or response.
  - *Usage Rate* ($\mathcal{UR}$) $= \frac{\#\mathcal{US}}{\#\mathcal{DS}}$ the relative amount of the used schemas out of the defined ones in each API. The goal of this indicator is to reflect on how many data schemas out of the total defined one are being actually used. Usage rates less than 100% indicate the presence of schema definitions which are not reachable from any operation.
- **API Data Model internal reuse**
  - *#Reused Schemas ($\#\mathcal{RS}$)*: The number of the schemas reused at least once. A reuse is detected when an embedded schema is redefined or referenced in another operation, or when a separately defined schemas is referenced more than once.
  - *Reuse Rate* ($\mathcal{RR}$) $= \frac{\#\mathcal{RS}}{\#\mathcal{US}}$ The number of schemas that are reused out of the total number the used schemas. The goal of this indicator is to assess the proportion of schemas reused at least once out of the used ones. Reuse rates of 0% indicate that every schema is used exactly once.
- **API Popularity**
  - *#Clones*: the number of exact duplicates found in different GitHub repositories. We consider that an API is popular if its specification is forked or pushed in different public repositories by many users on GitHub. For calculating it we count the number of time an identical clone is encountered in the database during the deduplication phase.

B. Metrics computation Example

In this section we explain the metrics computation through an example of a real-world API: BigQuery connection[1]. This API allows user to manage BigQuery connections to sources of external data.

Fig. 4 visualizes the API model of the BigQuery connection API, depicting both the structure and the data model of the API. The graphical notation used to depict the API's structure is detailed in [17], while the data model is represented as a simplified class diagram. This model is built based on the latest version (v1beta1) of the OAS description file of the API, updated on 21 August 2020.

BigQuery connection is a RESTful API, composed of six paths and eight operations. As in many other APIs, all paths start with the segment `/v1beta1` indicating the API version. Since this is the only path segment attached to the root of the API resource tree, the breadth is equal to one. The longest path (`/v1beta1/parent/connections`) is composed of three segments, thus the depth of the API is equal to three. The number of Operations Per Path (OPP) is 1.6, since two paths have more than one method. And, the API uses four distinct HTTP Methods.

In the case of the GET and the DELETE methods, there are only data schemas for the response message payloads, while for the POST and PATCH methods, schemas are used to define the content of both requests and responses. The specification of the API contains 17 Defined Schemas ($\#\mathcal{DS}$), which are all of the classes shown in Fig. 4. While 10 of them are the ones that are directly used by an operation, six are indirectly used by being referenced from another schema. Only one is not used, because it is neither referenced directly nor indirectly by an operation. (*Note that the `Not Used` data schema does not exist in the original API specification, it is only added for explanation purposes*). All schemas defined in the specification of this API are being used except the `Not Used` schema, so the the usage rate ($\mathcal{UR}$) is equal to $\frac{\#\mathcal{US}}{\#\mathcal{DS}} = 0.94$.

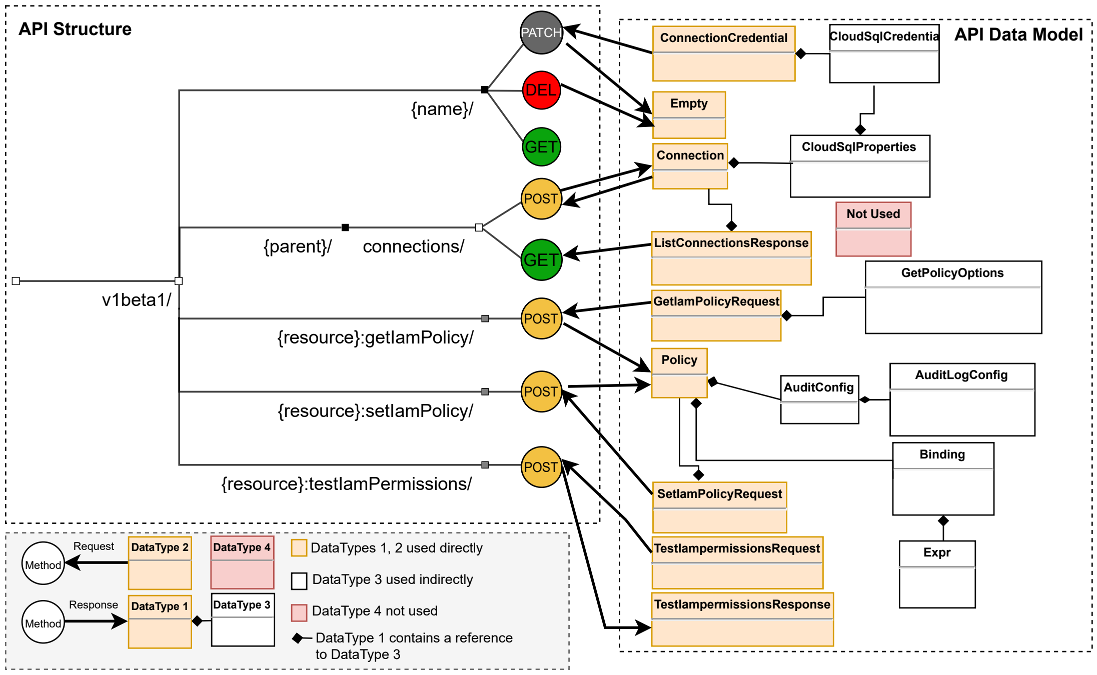
We can notice that a few of Data Schemas are being reused: `Connection`, `Policy`, `Empty`. Moreover, `Connection`, `Policy` have dependencies to other schemas `AuditConfig`, `AuditLogConfig`, `CloudSQLproperties`, `Binding`, `Exr`, and `CloudSqlConnection`. Thus, these schemas are also being indirectly reused, which results a number of nine reused schemas ($\mathcal{RS}$). This brings the reuse rate $\mathcal{RR} = \frac{\#\mathcal{RS}}{\#\mathcal{US}} = 0.56$.

In the specification file of this API all schemas are defined in the components section of the file (Listing 1). There are no schemas embedded in the request or response schema object of any operation.

Listing 1: Excerpt of the OAS description of the BigQuery connection, showing an example of schema definition in the components section

```
openapi: 3.0.0
servers:
  - url: 'https://bigqueryconnection.googleapis.
    ↪ com/'
info:
    ... Metadata about the API
paths:
    ... List of paths
```

[1]https://cloud.google.com/bigquery/docs/reference/bigqueryconnection/rest/



Metric	Value
#Paths	5
#Operations	8
Breadth	1
Depth	3
OPP	1.6
HTTP Methods	4
#Defined Schemas	17
#Used Schemas	16
#Reused Schemas	9
Usage Rate	94%
Reuse Rate	56%
Embedded Schemas	0%
#Clones	0

Fig. 4: Example: BigQuery API model and metrics

```
  '/v1beta1/{name}':
    delete:
      description: Deletes connection and associated credential.
      responses:
        '200':
          content:
            '*/*':
              schema:
                $ref: '#/components/schemas/Empty'
components:
  schemas:
    Binding:
      description: Associates 'members' with a 'role'.
        properties:
          condition:
            $ref: '#/components/schemas/Expr'
            description: The condition that is associated with this
              ↪ binding.
```

IV. Analytics Results

As we did for the example of BigQuery connection API (Figure 4), we computed the metrics for all 31,118 API specifications in our collection which make use of the `application/json` media type.

A. Statistics

To see how the data schemas usage in the APIs changes in function of the number of paths, we plot in Figure 5 a rough classification of the APIs depending on whether they feature zero, one or many paths and schema definitions. We also consider whether these schema definitions are used.

Figure 5 reveals the existence of 4266 APIs having no defined schemas in their specifications, but one or many paths, such as DropX API http://dropx.io/dropx-swagger.yaml provided by dropx.io and the Cisco PSIRT openVuln API https://api.cisco.com provided by cisco.com. They have respectively 7 and 20 paths, however no defined data schemas. Another small subset of the dataset (1536 APIs) are the API specifications having no listed paths but one or many

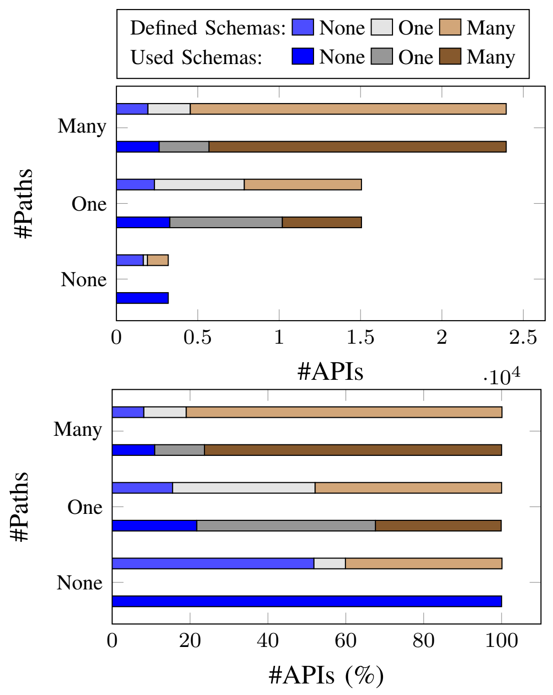


Fig. 5: API Classes: How many APIs have zero, one or many Paths? How many of those APIs have zero, one or many unique defined or used schemas?

	Min	Median	Mean	Stddev	Max
#Paths	1	2	3.98	±7.21	271
#Operations	0	3	5.23	±9.85	357
Breadth	1	1	1.97	±3.02	80
Depth	0	2	2.51	±1.80	19
Operations per path	0	1	1.30	±0.54	8
HTTP Methods	0	2	2.03	±1.23	8
#Defined Schemas	1	3	7.11	±18.69	580
#Used Schemas	0	2	4.42	±10.64	386
#Reused Schemas	0	1	1.47	±3.83	151
Usage Rate	0%	100%	77%	±31%	100%
Reuse Rate	0%	17%	31%	±36%	100%
Embedded schemas	0%	0%	27%	±40%	100%
#Clones	0	0	0.99	±9.65	459

TABLE I: Statistics for all metrics computed over 31118 APIs with media_types: application/json and #Paths$>$ 0 and #Defined Schemas$>$ 0

schemas. By manually verifying the software repositories of some specifications, we found out that some developers use the schema-only API specification for code generation.

While the relative majority of APIs (19418) have multiple paths and multiple schemas, there is a large number (12723) with a single path that has at least one schema definition. Finally, there are 1649 APIs which do not have any paths nor any schema definitions. These empty API descriptions have been excluded from the rest of the study as they do not describe any API structure nor any API data model.

Having delimited the sample in our study, in Table I we show the statistics for the metrics over the API specifications that represent request and response payloads using JSON and have at least one path (#Paths>0) and at least one defined schema ($\#\mathcal{DS} > 0$).

B. Correlation

To understand how the metrics we selected variate depending each on the other, we compute the correlation matrix shown in Table II. We observe a high correlation between the structural size metrics: number of paths and the number of operations found in an API (0.96). Also, the correlation between these two structure related metrics and the breadth is medium high (0.52, 0.60). There is also a similar correlation (0.58-0.63) between the structure metrics and the data model related metrics: the number of unique defined data schemas in an API's specification, the number of the actual used data schemas, and the number reused data schemas in the API.

In Figure 6 we show scatter plots visualizing the correlations between pairs of structural and data model size metrics with a high correlation, while in Figure 7 we show that the size in terms of paths is not linearly correlated with the number of clones of the API. The color of each dot on the density plot shows how many APIs share the same pair of metric values. To keep the color bar range in check, we have removed the highest density values close to the origin. The scatter plot also show a small number of large APIs with hundreds of paths and schemas that could be subject of further study.

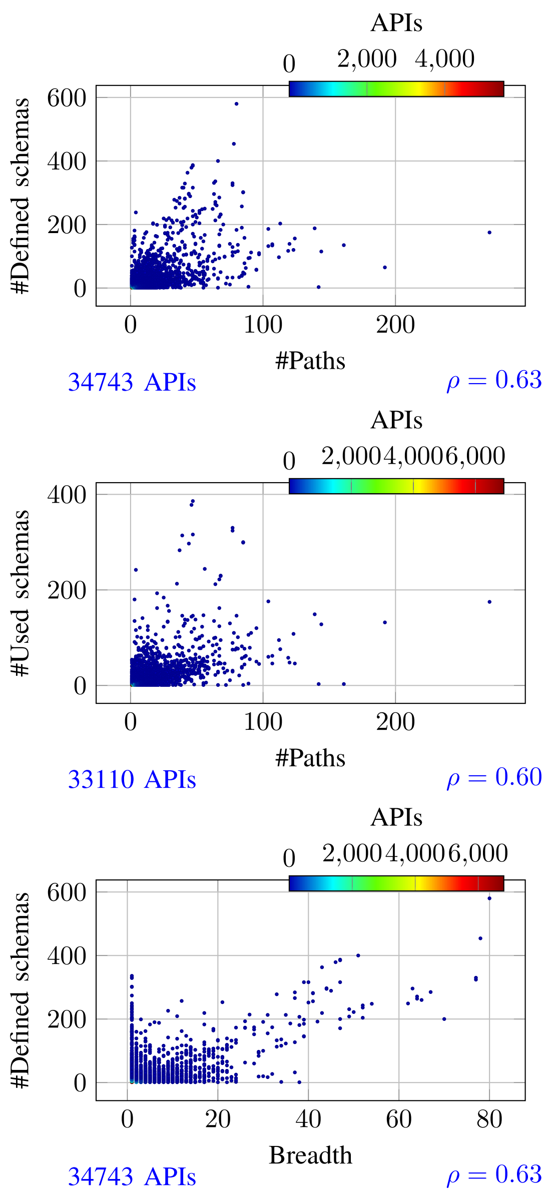


Fig. 6: Density scatter plots between #Paths and Breadth (X axis) and data model size metrics (Y axis). Each dot corresponds to one or more APIs having the same metric values.

C. API Styles Segmentation

In this section, we segment the large dataset to see how the metrics values depend on the specific group of APIs which provide different kinds of operations their clients.

To characterize different API architectural styles we distinguish which HTTP methods are found associated with their operations. We group all valid APIs according to the following classes:

(1) Read-only: APIs that only allow the client to read data with no provided operation to mutate the state of any resource.
(2) Read/Write: APIs that allow to the clients to perform both of the GET and POST operations on some resources.

	#Paths	#Operations	Breadth	Depth	OPP	Methods	#$\mathcal{DS}$	#$\mathcal{US}$	#$\mathcal{RS}$	$\mathcal{UR}$	$\mathcal{RR}$	$\mathcal{ES}$	#Clones
#Paths		●	●	•	·	•	●	●	●	·	·	·	·
#Operations	0.96		●	•	•	•	●	●	●	·	•	·	·
Breadth	0.60	0.52		·	·	·	●	●	•	·	·	·	·
Depth	0.27	0.28	-0.02		·	•	•	•	•	·	•	·	·
Operations per Path	0.01	0.16	-0.05	0.04		●		·	·	·	•	·	·
HTTP Methods	0.31	0.42	0.10	0.27	0.70		·	·	•	·	•	·	·
#Defined Schemas	0.63	0.58	0.63	0.17	-0.00	0.15		●	●	·	·	·	·
#Used Schemas	0.60	0.58	0.51	0.16	0.02	0.17	0.78		●	·	·	·	·
#Reused Schemas	0.59	0.60	0.44	0.18	0.10	0.26	0.66	0.86		·	•	·	·
Usage Rate	-0.05	-0.03	-0.05	-0.10	0.06	0.03	-0.17	0.05	0.06		·	•	·
Reuse Rate	0.12	0.16	0.06	0.11	0.33	0.37	0.01	0.02	0.26	0.10		·	
Embedded Schemas	-0.05	-0.05	-0.02	-0.07	-0.07	-0.09	-0.10	-0.08	-0.11	0.24	-0.12		·
#Clones	-0.02	-0.02	-0.02	-0.02	-0.01	-0.02	-0.01	-0.01	-0.01	-0.01	-0.00	-0.01	

TABLE II: Metrics correlations for 31118 APIs with media_types: application/json and #Paths > 0 and #DS > 0

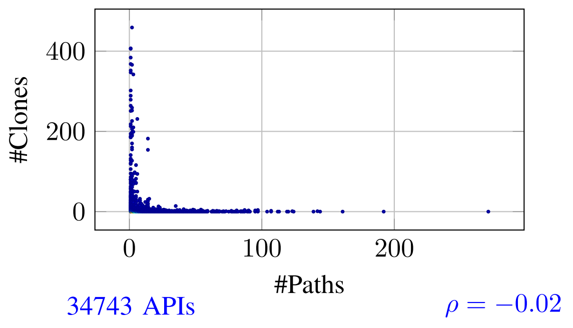


Fig. 7: Scatter plot with the number of clones vs. the number of paths in APIs with at least one clone

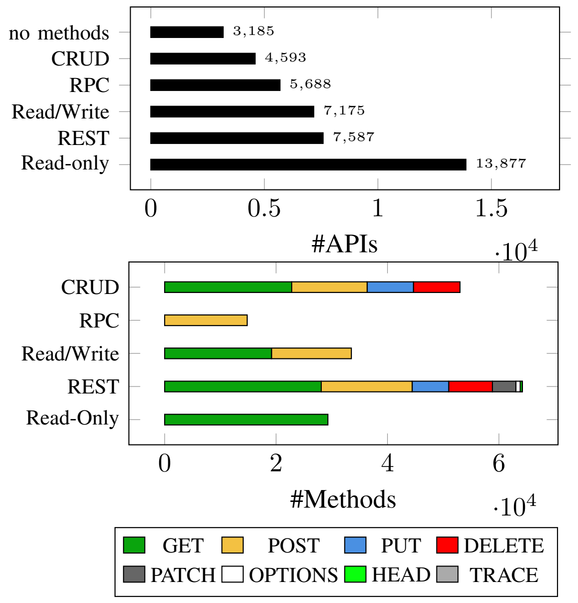


Fig. 8: APIs clustered by HTTP method combination

(3) RPC: APIs that allows the client to interact with remote resources by means of the POST operation only.

(4) CRUD: APIs that allow clients to invoke only the Create, Read, Update and Delete operations using respectively the POST, GET, PUT and DELETE methods.

(5) No methods: API specifications that have no HTTP methods listed within their paths.

(6) REST: The rest of the APIs that use any of the other 68 combinations of HTTP methods.

In Figure 8 we classify the 42194 APIs with valid specification and compute the total number of methods for each class. And, in Figure 9 we visualize how the API styles are related to the API size by computing the total number of APIs having a given style for each value of the #Paths (up to 10). We see that most of the small APIs offering only one path are Read-only or RPC-style APIs. These styles appear less in the case of larger APIs, where the most dominant styles are the CRUD and REST.

In Figure 10, we show the data schema sizes for each APIs style. The top bar chart visualizes for each style how many used schemas are present (zero, one, or more). The bottom bar chart shows whether schemas are defined embedded within the operation descriptions or, as in most cases, in a separate section of the API specification. The proportion of embedded schemas varies between 10% (REST) and 18% (Read-Only).

V. Discussion

A. How big are the API models?

According to the computed metrics, we found 17 APIs with a number of paths that goes up to more than 100; The largest RPC-style API provides 271 operations, all accessed using the POST method; The API with the highest number of operations (357) is a read/write API, with only GET and POST methods. The largest API data model we found includes 580 distinct schema definitions.

These are exceptionally large sizes, as 12723 APIs have only one path and, in one case, up to 70 unique defined schemas, out of which only 1 is actually used. Another single path API with only one POST method describes the payload of the request and multiple responses using 30 distinct schema types.

B. Does the API style affect the API size?

We discovered that the size of the APIs is affected by their styles. Most of the smallest APIs (in term of number

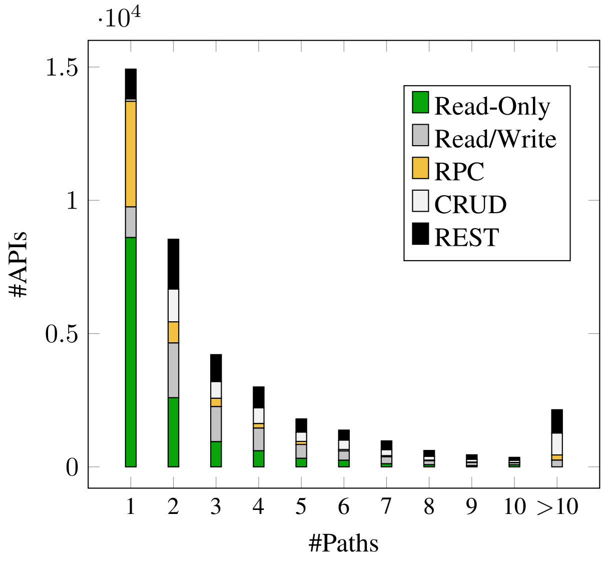
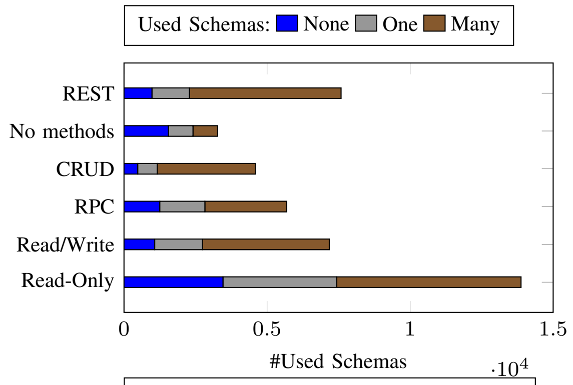


Fig. 9: Styles of the APIs depending on #Paths



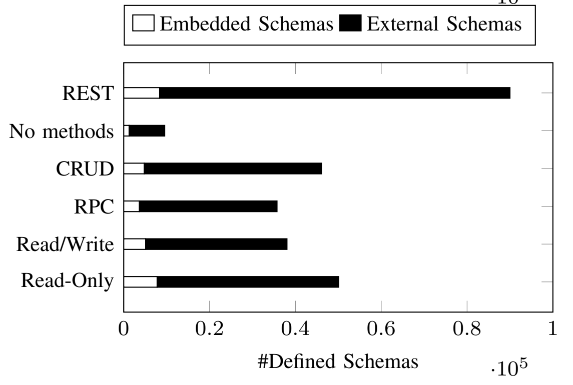


Fig. 10: Data Model sizes for each API style

of paths) are Read-Only and RPC styled APIs. In the case of APIs having more than two paths, they tend to have various combinations of HTTP methods. The CRUD and REST styles are also the ones whose APIs overall include the highest total number of HTTP methods.

C. What is the relationship between the functional structure and the data structure of an API?

We have seen that there exist API descriptions that only enumerate data model schemas without any resource paths or operations and vice versa. For APIs including both, there is a medium-high correlation between the size of the data model and the number of paths and operations.

More than half of the non-empty API specifications made full usage of the defined schemas; only very few models did not contain any reference to the schema definitions from the API operation descriptions. 81.1% of the 23943 APIs that are composed of at least two paths contain also at least two defined Schemas. The Usage Rate in these APIs reaches 76.3%. This can be seen in Fig. 5 where we can see how in the size and usage in API data models in function of number of paths.

D. How developers (re)use schemas within the same API?

We found that most on the APIs use references to externally defined schemas instead of including embedded ones. This facilitates data schemas reuse within the same API. Still, 25% of the total number of APIs in our collection contains at least one schema embedded in either the request or response descriptions. Not all of the analysed APIs had some reuse cases, however, we found a case where the number of distinct reused data schemas goes up to 151.

VI. Threats to Validity

External Validity - Sampling Bias The results of this study have been obtained by measuring APIs specifications found in public open source repositories. Although we have filtered duplicates and invalid specifications, the results may not be representatives of APIs described using other meta-models or APIs meant for internal use only.

VII. Related Work

Analysing Web APIs through their human-readable description models is a practice followed by many empirical studies [11, 13, 5, 20].

OpenAPI descriptions have been analyzed for studying the evolution of web APIs and its impacts on the clients. In [10], we focused on observing how the size of web APIs, simply defined in terms of the number of operations, changes during the API evolution lifecycle. The authors of [21], use the API model written in OAS to determine whether developers inform their clients about deprecation during the API's evolution.

In this work, we do not take into account the time dimension as we only focus on the latest version of the OAS descriptions. We also consider a larger set of metrics reflecting more details about the structural aspect and the APIs data model. In their work [5], Haupt et al. performed a structural analysis of REST APIs, for supporting API governance tasks, on 286 API description documents (RAML and Swagger) retrieved from https://apis.guru, considering only metrics related to the APIs structures: number of resources and HTTP methods, the depth and the breadth of the API structure, which are also included in our study. They observed that APIs do not take advantage of the full power of REST (especially the HATEOAS concept) [5]. In fact, like OpenAPI, also Swagger and RAML provide no explicit means for describing hyperlinks between resources and they have no support to describe the relationship between POST requests on one resource and the resulting creation of another one. They discovered that APIs in their dataset are on average small with a median of 9 resources per API. An additional observation confirmed by our larger study is that read-only resources are very common and a subset of APIs are completely read-only. The authors validate also their derived metrics for the user-perceived complexity of APIs, with a survey among API designers and developers. While all the analyzed API descriptions in [5] are selected from a curated repository. In our work we consider a larger set of specifications from different public repositories on GitHub, including the latest API.guru collection.

OAS API models were also study subjects for structural patterns mining works. In [17], Serbout et al. presented a pattern mining approach applied on a collection of 6,919 Swagger and OpenAPI descriptions. These are fed to a model from which API structure trees can be built and later, fragmented in order to detect APIs with reoccurring structures, which were classified into pattern primitives and design smells.

Wittern et al. studied GraphQL APIs by analyzing 16 commercial GraphQL schemas and 8,399 GraphQL schemas mined from GitHub projects. They analyzed data types and the possible operations and proposed a characterization of naming conventions that can help developers to adopt community standards to improve API usability. Furthermore, they detected that the majority of GraphQL APIs are susceptible to denial of service attacks through complex queries [20].

VIII. Conclusion

In this study we analyze the functional structure and the data model of Web APIs, a key element of microservices and service-oriented architectures, to explore and describe a large API collection to better understand API design practices. We defined multiple metrics and indicators to quantify the size of Web APIs described using the OpenAPI standard interface description language. We computed the metrics over a large collection of 42194 valid Web APIs mined from public repositories on GitHub. We present descriptive statistics which indicate that the median and mean sizes are small (e.g., 3.98 paths, 5.23 operations and 7.11 defined schemas). Still, some APIs in the collection include up to 271 paths, 357 operations and 580 defined schemas. The API size varies depending on the chosen sample, as we have shown by grouping APIs depending on the types of operations they offer. In addition to measuring both API structures and data models, we observed that their size is fairly correlated. Within the same API, modelers also tend to reuse schema definitions, which are mostly referenced from the request and response descriptions.

References

[1] Fernando López de la Mora and Sarah Nadi. An empirical study of metric-based comparisons of software libraries. In *Proc. 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, page 22–31, 2018.

[2] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. OpenAPItoUML: A Tool to Generate UML Models from OpenAPI Definitions. In *Proc. International Conference on Web Engineering (ICWE)*, pages 487–491, 2018.

[3] Matthias Galster and Danny Weyns. Empirical research in software architecture: How far have we come? In *Proc. WICSA*, pages 11–20, 2016.

[4] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. An analysis of restful apis offerings in the industry. In *Proc. International Conference on Service-Oriented Computing (ICSOC)*, pages 589–604, 2017.

[5] Florian Haupt, Frank Leymann, and Karolina Vukojevic-Haupt. API governance support through the structural analysis of REST APIs. *Computer Science - Research and Development*, 33(3-4): 291–303, aug 2018.

[6] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *Proc. ICST*, pages 131–141, 2020.

[7] István Koren and Ralf Klamma. The exploitation of OpenAPI documentation for the generation of web frontends. In *Companion Proc. The Web Conference*, pages 781–787, 2018.

[8] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. A data-driven approach to measure the usability of Web APIs. In *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 64–71, 2020.

[9] Arnaud Lauret. *The Design of Web API*. Manning, 2019.

[10] Fabio Di Lauro, Souhaila Serbout, and Cesare Pautasso. Towards large-scale empirical assessment of Web APIs evolution. In *Proc. ICWE*, May 2021.

[11] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating web apis on the world wide web. In *Proc. 8th IEEE European Conference on Web Services (ECOWS)*, pages 107–114, 2010.

[12] Leonard Richardson Mike Amundsen, Sam Ruby. *RESTful Web APIs*. O'Reilly, 2013.

[13] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An analysis of public rest web service apis. *IEEE Transactions on Services Computing*, 14(4):957–970, 2021.

[14] Sam Newman. *Building microservices*. O'Reilly, 2015.

[15] Martin P Robillard and Robert DeLine. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[16] Carlos Rodríguez et al. REST APIs: A large-scale analysis of compliance with principles and best practices. In *Proc. ICWE*, pages 21–39, 2016.

[17] Souhaila Serbout, Cesare Pautasso, Uwe Zdun, and Olaf Zimmermann. From OpenAPI fragments to api pattern primitives and design smells. In *Proc. European Conference on Pattern Languages of Programs (EuroPLoP)*, 2021.

[18] The Open API Initiative. OAI. https://openapis.org, 2021.

[19] Fredy H Vera-Rivera, Carlos Gaona, and Hernán Astudillo. Defining and measuring microservice granularity—a literature overview. *PeerJ Computer Science*, 7:e695, 2021.

[20] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart, and Louis Mandel. An empirical study of GraphQL schemas. In *Proc. ICSOC*, pages 3–19, 2019.

[21] Jerin Yasmin, Yuan Tian, and Jinqiu Yang. A first look at the deprecation of RESTful APIs: An empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 151–161. IEEE, 2020.

[22] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding architectural decision making on quality aspects in microservice apis. In *Proc. ICSOC*, pages 73–89, 2018.