How Composable is the Web? An Empirical Study on OpenAPI Data model Compatibility

1st Souhaila Serbout
Software Institute (USI)
Lugano, Switzerland
souhaila.serbout@usi.ch

2nd Cesare Pautasso
Software Institute (USI)
Lugano, Switzerland
c.pautasso@ieee.org

3rd Uwe Zdun
University of Vienna
Faculty of Computer Science, Austria
uwe.zdun@univie.ac.at

***Abstract*—Composing Web APIs is a widely adopted practice by developers to speed up the development process of complex Web applications, mashups, and data processing pipelines. However, since most publicly available APIs are built independently of each other, developers often need to invest their efforts in solving incompatibility issues by writing ad-hoc glue code, adapters and message translation mappings. How likely are Web APIs to be directly composable?**

The paper presents an empirical study to determine the potential composability of a large collection of 20,587 public Web APIs by verifying their schemas' compatibility. We define three levels of data model elements compatibility – considering matches between property names and/or data types – which can be determined statically based on API descriptions conforming to the OpenAPI specification. The study research questions address: to which extent are Web APIs compatible; the average number of compatible endpoints within each API; the likelihood of finding two APIs with at least one pair of compatible endpoints.

To perform the analysis we developed a compatibility checker tool which can statically determine API schema compatibility on the three levels and find matching pairs of API responses which can be directly forwarded as requests to the same or other APIs. We run the tool on a dataset of 751,390 request and response message schemas extracted from publicly available OpenAPI descriptions.

The results indicate a relatively high number of compatible APIs when matching their data models only on the level of their elements' data type. However, this number gets lower narrowing the scope to only the ones handling data objects having identical properties name. The average likelihood of finding two compatible APIs with both matching property names and data types reaches 21%. Also, the number of compatible endpoints within the same API is very low.

***Index Terms*—Web APIs, OpenAPI Specification, API Composabilty, Endpoints Composability, Empirical Study.**

I. Introduction

Service providers rely on mechanisms such as HTTP APIs to expose their services to multiple clients through public endpoints of different structures [18] on the Web [3]. Very often, they choose JSON as the main data interchange format [17]. To speed up the implementation of client applications, developers rely on Mashup integration and discovery tools and follow specific approaches [13], [14], [24] to minimise their efforts of finding API candidates that can be chained or aggregated [7], [16].

APIs operations and their data models can be described using Interface Description Languages. The currently most widespread one among developers is the OpenAPI Specification [20] (which we call henceforth for brevity: OAS). It is an internal Domain Specific Language (DSL) of the JSON data serialization format. The language uses an adapted version of JSON Schema [2] to describe the data model of Web APIs. Nowadays, there are many tools built around OAS, facilitating APIs descriptions parsing, validation, documentation, and code generation. However, there exists none for systematically comparing and statically checking the compatibility of schemas describing the data communicated by different API endpoints.

In this work, we developed a static matcher of source APIs responses schemas and sink APIs request schemas, in order to discover the ones with compatible endpoints. The tool can be used interactively by Mashup developers: given a response obtained from an API, it can provide recommendations about compatible API requests.

To compare the schemas, we define different types of matching strategies and we apply them to determine schemas compatibility on three levels (property names and/or data types) and measure their impact on whether messages produced by a given source API can be directly consumed by another sink API. While these detected matches can be provided to Mashup developers as part of auto-completion and API recommendation tools [8], in this paper, we take a broader view and empirically analyze the overall compatibility of a large collection of real-world Web APIs.

Within this context, we attempt to answer the following research questions:

Q1: *How compatible are Web APIs?*
Our goal is to evaluate the level of compatibility of Web APIs' data models described using OAS, starting from a large representative sample. APIs with compatible schemas are potentially directly composable, i.e., they can be used to build a Mashup without the need to introduce any complex adaptation. We consider directly composable also APIs that

The work is funded by the SNF, with the API-ACE project nr. 184692.

require simple adaptations, such as obvious data types conversion (e.g: number to string).
We do not assume that any pair of APIs with compatible data model elements should be used in a purposeful Mashup, as the underlying semantic meaning of schemas having the same types and property names can be vastly different. Before deciding to compose APIs with compatible data model elements, Mashup developers usually take into account additional knowledge available in higher-level semantics information present in natural language descriptions that are not considered in this study. However, when APIs provide or consume schemas that are compatible, this ensures that no data representation mismatches will occur when attempting to compose them.

Q2: *On which level are most API compatible?* Strict schemas compatibility requires both schema elements data types and objects' properties names to match. This ensures that JSON payloads transferred from one API to the next will not break the corresponding schema compliance checking usually found at the API boundary. In this study, we relax this requirement and analyze both object property name and data type matches independently so that we can quantify the impact of each dimension on the compatibility of each pair of API endpoints. Property name matches indicate potential semantic matches that would require some data type conversion when composing the corresponding endpoints. Looking for compatible data types focuses on syntactic matches, which could lead to composable endpoints by finding a suitable parameter name mapping.

Q3: *What is the likelihood that a random pair of APIs has at least one compatible endpoint?* A compatible pair of endpoints is composed of a sink and a source endpoints handling data resource of compatible schemas. Answering this question for different segments of our sample of Web APIs could help to generalize the results to other Web APIs collections. We study whether and how such probability depends on the API size, defined as the number of endpoints.

Q4: *To what extent are the APIs endpoints compatible?* While we consider "compatible" a pair of APIs having at least one compatible pair of endpoints, we are interested to determine to which extent all of the endpoints of an API can be compatible with some or all the endpoints of other APIs.

The results show that the number of compatible APIs in the analyzed collection varies depending on the chosen matching level (between 15 million pairs of APIs with compatible data model elements on the Data-Type Level and approx. 0.5 million pairs of APIs when considering both Data-Type and Property-Name). Most compatible pairs of endpoints are found across different APIs. There is a larger proportion of compatible APIs which can act as a source as opposed to a sink, although the majority can play both roles. Within each API that is compatible with another one from the studied collection, there is a median of 3-5 compatible endpoints, leading to an average rates of 18% endpoints potentially composable as sinks and 28% as sources when using the strictest matching level that considers both data types an property names. Overall, the probability of random pairs of APIs being compatible tends to grow with the size of the API and on average is 21%.

The rest of this paper is structured as follows. We present the dataset and give some background on OAS data models in Section II. We present the schema compatibility analysis method in Section III and illustrate and discuss the results in Sections IV and V. We summarize the related work in Section VII and draw some conclusions in Section VIII.

II. Background: Dataset and OAS Schemas

A. APIs Dataset Overview

1) Dataset Collection: The dataset used in this study is composed of de-duplicated OAS descriptions of 106,873 APIs: 56,536 was collected directly from public software repositories in Github, using the Github API, and the rest was obtained from a dataset shared by Assetnote in the context of BSides Canberra 2021 conference[1].

Our Github search query targeted YAML/JSON files that contain some OAS required properties. Any valid OAS description is required to contain a field `OAS` or `swagger` which specifies the version of the language used in the current specification. It is also required to have an `info` section, which should be mandatory to have a `title`. These specification details are the main heuristics based on which our Github search query was built.

An API description can be written all in one file, as it can be split over several files linked through references using the `$ref` JSON schema construct. For that reason, while downloading the found potential OAS descriptions, we extracted all the external references that are pointing to other files, followed them, and downloaded them. However, we could not download the referenced files in the case of the descriptions we collected from the dataset shared by Assetnote, as they did not share the sources where each file comes from.

2) Dataset Preparation: We extracted from that collected dataset the most realistic API descriptions: First, we filtered valid descriptions, then we selected the ones that have at least one realistic server (OAS v3) and base path (Swagger) URL. This ensures the dataset includes only specifications describing APIs of services with valid endpoints. We also selected APIs that have at least one endpoint with a description of schema of consumed or produced data, and have at least three paths in order not to bias the average composability rate. The resulting collection we analyze in this study is composed of 20,587 distinct and valid API descriptions, with 583,207 response schemas and 168,183 request schemas, leading to potentially 98,085,502,881 schemas matches (Table I).

B. Comparable Schema Objects in OpenAPI Specification

In OAS, responses and requests data are described through defining Schema Objects which uses 16 properties that are directly taken from the JSON Schema specification, which include: `title`, `enum`, `required`, `minimum`, `maximum`, `multipleOf`, `minItems`, `maxItems`. We denote these properties set $\mathcal{P}_{\mathcal{J}}$. Other 11 keys are also taken from the JSON

[1]https://blog.assetnote.io/2021/04/05/contextual-content-discovery/

Schema specification, but they are adapted to the Web APIs data models' context, including: `type`, `format`, `oneOf`, `anyOf`, `allOf`, etc. We denote this set of keys $\mathcal{P_A}$.

As for JSON Schema, OAS supports describing data of primitive data types: *integer, string, number, boolean*, and data of complex nested structures which are the *arrays* and the *objects*. An *array* is a sequence of items that can be of primitive or complex data types. And an *object* is a map of key-values, where the values can be of primitive or complex data types. These data types define the structure of the JSON object that is communicated as part of the HTTP response or request body exchanged with a specific API endpoint.

Note that in OAS a valid specification cannot contain invalid JSON schemas.

Given the wide variety of schema descriptions found in the collection, we formalize a valid and statically comparable Schema Object in OAS as a JSON object that conforms to the following rules:

Let $\mathcal{S}$ be a valid schema, $\mathcal{S} = \bigcup_{i \in I}(k_i, value(k_i))$, $I = \{i \in \mathbb{N}, i \leq n\}$, where n is the total number of top-level keys of $\mathcal{S}$.

The possible values type of each key $k \in \mathcal{P_J} \cup \mathcal{P_A}$ are:

$$type(value(k)) = \begin{cases} number & \text{for} \quad k \in \mathcal{P_J} \setminus \{title, pattern, \\ & \qquad required, enum\} \\ string & \text{for} \quad k \in \{title, pattern, type, \\ & \qquad format, description\} \\ simple\_array & \text{for} \quad k \in \{required, enum\} \\ complex\_array & \text{for} \quad k \in \mathcal{P_A} \setminus \{type, format, \\ & \qquad description\} \\ object & \text{for} \quad k \in \{properties, items\} \end{cases}$$

A $simple\_array$ is an array of strings or numbers, while $complex\_array$ is a complex array of JSON objects or arrays. In the case of the $pattern$ property, the value should be a valid regular expression.

APIs with schemas that do not conform with this validity rule do not make it possible to statically determine whether they are compatible or not. Therefore $59,593$ schemas have been excluded from the comparison.

C. Extracted Schemas Overview

To extract the data schemas descriptions, we parsed all the API endpoints responses and request bodies. Our schemas compatibility checker supports both Swagger 2.0 and OAS 3.0 schemas. In an API description written in OAS, the data schemas can be found in the section `definitions` in the case of the Swagger 2.0 or in the section `components` in the case of OAS 3.0 and 3.1, then referenced using the `$ref` JSON construct (e.g: `{"$ref":'/path-to-schema'}`) in the endpoint where it is used. In this case, according to OAS terminology, the schema is used as a Reference Object, that can be internal or external. A data schema can also be defined in its usage place. To simplify the APIs description and reduce schemas extraction task complexity, we first bundle the description split over several files into one and resolve all the internal references. The resulting description file does not contain any `$ref` key. Then we proceed in extracting

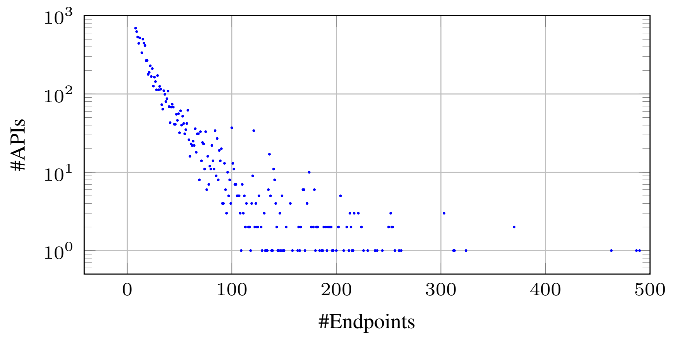


Figure 1. API Size Distribution

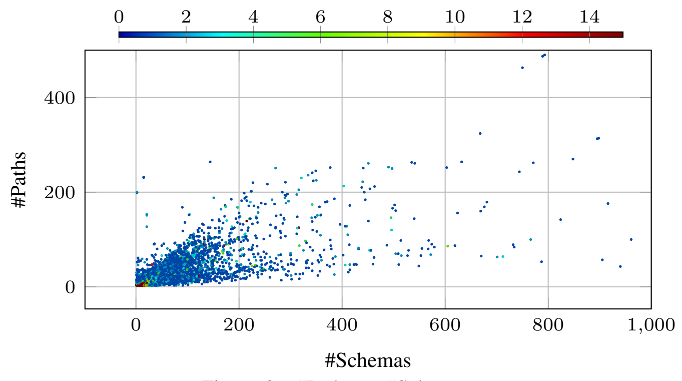


Figure 2. #Paths vs #Schemas

the schemas of each endpoint. Note that schemas with cyclic references are not extracted.

From 20,587 APIs we extracted a total of 751,390 valid Data Schemas, where 77.6% are request schemas (we note this set of schemas $\mathcal{S_{REQ}}$) and 22.4% are associated with responses ($\mathcal{S_{RES}}$). Table I shows that there are at least three times more responses than requests. This is due to presence of many GET operations, which only provide a response schema (Table II).

In Fig. 1 we show how many APIs have the same number of endpoints. In Fig. 2 we show the correlation between API Size (the #Paths) and the total number of extracted data schemas for each API. The density scatter plots use color to indicate how many APIs share the same metric values.

The #Paths indicates the size of the API structure [17], while the #Endpoints measures how many paths refer to at least one schema which can be statically compared.

D. Complex Operators Usage

The OAS language inherits from JSON Schema the possibility of defining constraints over sub-schema combinations using the `allOf`, `oneOf` and `anyOf`. Schemas become more complex to statically analyse when they contain nested data structures or use composition.The `allOf` construct is used to define a data structure that is the union of the data structures defined by the sub-schemas. The `oneOf` construct picks one of the sub-schemas. The `anyOf` construct defines the union of some sub-schema combination. In Table III we show how often these constructs are used in our dataset. Because of its

Table I
EXTRACTED VALID SCHEMAS OVERVIEW

	Number of Schema Types			Number of Schema Types per API						
	Total	**Primitive**	**Complex**	**Min**	**Avg**	**Max**	**StdDev**	**Median**	**Q1**	**Q3**
$\mathcal{S}_{\mathcal{RES}}$	583,207	66,765	516,442	0	30.22	2,513	75.97	12	6	28
$\mathcal{S}_{\mathcal{REQ}}$	168,183	5,369	162,814	0	8.19	928	17.36	3	1	8
$\mathcal{S}_{\mathcal{RES}} \cup \mathcal{S}_{\mathcal{REQ}}$	751,390	72,134	679,256	1	19.20	2,513	53.33	8	5	18

Table II
HTTP METHODS USAGE IN THE STUDIED APIS

Method	**Min**	**Avg**	**Max**	**Median**	**Q1**	**Q3**
GET	0	10.60	1560	5	3	11
POST	0	8.40	564	4	1	8
PUT	0	2.04	344	1	0	2
DELETE	0	1.82	344	1	0	2
PATCH	0	0.92	344	0	0	0

Table III
USAGE OF THE `allOf`, `oneOf`, `anyOne` AND `not` OPERATORS IN THE API SCHEMAS

	#Constructs		#Schemas		
Operator	$\mathcal{S}_{\mathcal{RES}}$	$\mathcal{S}_{\mathcal{REQ}}$	$\mathcal{S}_{\mathcal{RES}}$	$\mathcal{S}_{\mathcal{REQ}}$	$\mathcal{S}_{\mathcal{RES}} \cup \mathcal{S}_{\mathcal{REQ}}$
allOf	181,714	110,494	140,438	47,143	187,581
oneOf	33,975	11,925	28,442	11,691	40,133
anyOf	5,991	3,444	5,950	3,438	9,388
not	153	81	29	28	57

low usage (Tab. III) and complex semantics [4], in this study we excluded schemas using a `not` operator (already filtered in Table I).

III. API SCHEMA COMPATIBILITY ANALYSIS

Our compatibility analysis is based on searching for matching APIs' data models elements, to statically determine whether data retrieved from an API endpoint can be directly sent to another one. The result is a classification of the APIs and their endpoints as follows:

Definition 3.1 (Endpoint Match): A pair of endpoints belonging to the same or different APIs where the source endpoint has at least one response schema that is compatible (on a certain level) with a request schema of a sink endpoint.

Definition 3.2 (API Match): A pair of APIs with at least one endpoint match.

Definition 3.3 (Compatible API): An API with at least one endpoint involved in an endpoint match.

For example, for each API response obtained from a GET (or any other HTTP method) request we search for a matching JSON structure described in the schema of the request body of another endpoint (with a POST, PUT, PATCH method) in the same or other APIs in the collection. To do so, we detect matches based on three compatibility relationships (*matching levels*) between the pair of distinct endpoints.

Definition 3.4 (Data-Type Match): A pair of schemas with identical element data types.

The first matching strategy is applicable to schemas with both complex and primitive types. It focuses on ensuring that the request message content can be built from the data structures found in the responses, but – in case of complex data types – it may require a mapping between data elements of compatible primitive types since it ignores their names.

Definition 3.5 (Property-Name Match): A pair of schemas with identical element names.

The goal of the second level is to compare two schemas taking only into account the semantic knowledge they contain. Responses which semantically match requests can be forwarded between APIs, assuming that suitable data type conversion can be introduced. This level concerns only complex datatypes: `objects` and `arrays of objects`, as they are the ones that hold in the names of their properties concise semantic knowledge about the described data.

Definition 3.6 (Property-Name and Data-Type Level Match): A pair of schemas with identical element names and types.

This is the most strict schema compatibility level, which detects responses which can be directly be forwarded as requests without any fields renaming or data type conversion. While doing so will not break the schema conformance checking usually found on the sink API side, it may not always be meaningful to compose such APIs in a useful Mashup application.

A. Impact on Composability

The implication between compatible schemas and composable endpoints does not always hold. As shown in Figure 3, it may be possible to compose incompatible schemas by introducing suitable adapters. Likewise, it may be impossible to compose compatible endpoints due to higher-level semantic mismatches.

We distinguish:

Definition 3.7 (Directly Composable Match): A pair of schemas with identical element names that are also syntactically compatible with identical data types or where **all** sink types are strings.

These matches can be potentially lead to directly composable endpoints, whose responses that can be forwarded without any translation to the next API.

Definition 3.8 (Indirectly Composable Match): We distinguish two situations:

*S*1: A pair of schemas with **identical data types**, for which a mapping between mismatching property names can be provided to translate the responses into compatible requests. The identical data types make it possible to attempt to rename the corresponding property names to perform a simple message translation.

*S*2: A pair of schemas with **identical schemas elements name** makes it also possible to attempt data types adaptation, in case of data types mismatch. However, in some cases the data types conversion can be impossible.

Definition 3.9 (Not Composable Match): A pair of schemas with identical element names, where all source and sink types are different and no data type conversion is possible.

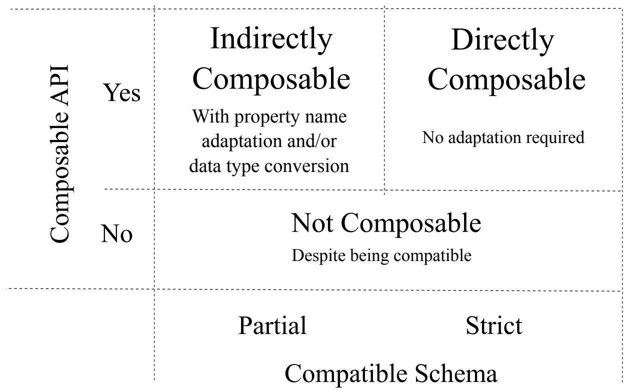


Figure 3. Relationship between API Compatibility and Composability

In general, Property-Name Level matches can be composable if it is possible to resolve the Data-Type Level mismatches. For example, it is possible to convert any primitive type into a string. While the previous definitions can be generalized to pairs of schemas where trivial conversions are possible between all, some or no pair of types, in this paper we further classified Property-Name Level matches according to the previous string-based definitions.

We also do not consider cases in which from a large response a subset of the data can be extracted and forwarded as a smaller request to another API.

B. Schema Matching Algorithm

In this section we explain the approach followed to detect compatibility between data schemas of two endpoints on three different levels: Data-Type, Property-Name, Data-Type & Property-Name Level.

Given the different representations of equivalent schemas (an example is shown in Fig. 4), we perform different schema pre-processing steps. These steps depend on the type of relationship we want to detect, and they aim at bringing all the different schemas descriptions into a canonical form, which makes the compatibility check more efficient.

Property-Name Level This comparison aims at matching the semantic data models of response payloads and requests data structures of two different endpoints.

For instance, the Schemas ❶, ❷ and ❸ are both describing objects with properties of the same name, however the objects representations are described differently. An instance of Schemas ❶ and ❷ are also valid against ❹. However, objects that have in addition the `email` property are also valid against ❹, making the matches between ❹ and each of the ❶ and ❷ only a possible one.

If two schemas describe objects or arrays of objects having the same properties names this implies also their structural similarity, however, they are not necessarily syntactically compatible due to mismatches of the primitive types of the object or array properties with matching names. We further study one case in which the sink schema uses string types, which can receive data from any other primitive type with straightforward data conversion. Other simple data type conversions may be possible to ensure the composability of endpoints that match on a Property-Name Level only.

Data-Type In this matching level, we are interested in only comparing the schemas' data types. While the schemas of primitive data types are easy to compare, the complex ones require schemas simplifications and abstraction to be able to compare them. First, the compared schema definitions should be structurally identical, then the data types of all their properties should be also identical to be considered "compatible". The Schemas ❶, ❷ and ❸ are compatible also in this matching level.

C. Canonical Form Transformation

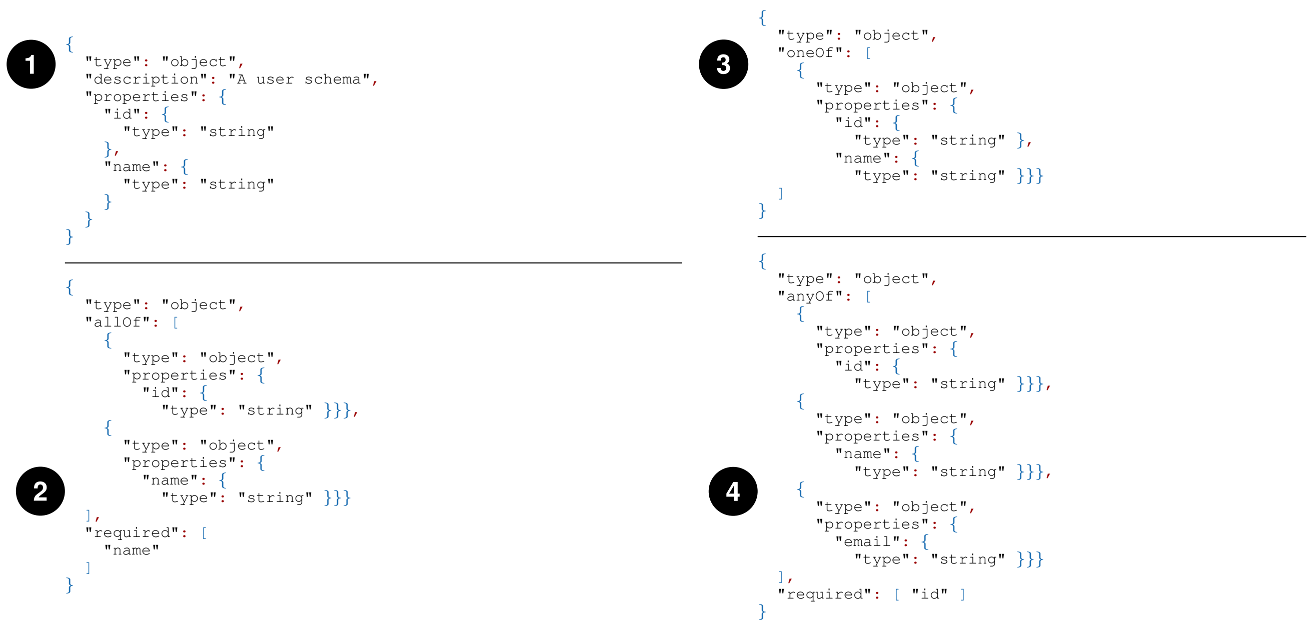
The richness of the JSON format makes it not trivial to identify the relationship between two JSON Schemas. Equivalent schemas can be described in very different ways (Fig. 4). The goal of the canonical transformation is to obtain a unified schema representation that can be efficiently compared for each matching level. We show an example in Fig. 5. The canonical form used to compare the schemas depends on the matching level and on whether the schema is of a complex or a primitive data type (Fig. 6). The transformation uses the following steps:

1) **Irrelevant properties removal.** In the OAS standard the following schema properties (`description`, `title`, `default`, `example`, `readOnly`, `writeOnly`, `example`, as well as the $\mathcal{P}_{\mathcal{J}}$ properties listed in Section II-B). These properties are not relevant for the matching process so they are removed from the schemas. This step is also important to reduce the schemas comparison cost.

2) **Complex operators simplification**. The complex operators `allOf,` `anyOf` and `oneOf` are simplified in order to be able to compare the data schemas on each matching level. *allOf.* If a schema $S = \bigcup_{i \in I}(k_i, value(k_i))$ is using an `allOf` operator, this mean that there exist a key k_i for which $value(k_i)$ is an array of sub-schemas $s_1, s_2, .., s_m$. The simplified schema S is a schema that merges all the properties of the sub-schemas $s_1, s_2, .., s_m$ without the `allOf` operator.
oneOf. The schema is simplified by creating a new schema for every sub-schema found within the `oneOf` entries.
anyOf. The schema is simplified to an array of all the possible schemas combinations obtained by recombining all schemas from the array under the *anyOf* element.

3) **Schemas properties sorting.** The goal of this step is to unify the structures of the data schemas that should be compared on a syntactic level only. First we sort each nested Schemas Object by the number of nested Schemas Objects it contains, then by Number of nodes and finally by their depths. And finally, we sort all the nested Schema Objects alphabetically based their types.

4) **Schemas properties labels abstraction.** After unifying the schema's structure, we replace the labels by abstract ones so that two schemas having the same set of types have an identical description.



```
{
  "type": "object",
  "description": "A user schema",
  "properties": {
    "id": {
      "type": "string"
    },
    "name": {
      "type": "string"
    }
  }
}
```

```
{
  "type": "object",
  "allOf": [
    {
      "type": "object",
      "properties": {
        "id": {
          "type": "string" }}},
    {
      "type": "object",
      "properties": {
        "name": {
          "type": "string" }}}
  ],
  "required": [
    "name"
  ]
}
```

```
{
  "type": "object",
  "oneOf": [
    {
      "type": "object",
      "properties": {
        "id": {
          "type": "string" },
        "name": {
          "type": "string" }}}
  ]
}
```

```
{
  "type": "object",
  "anyOf": [
    {
      "type": "object",
      "properties": {
        "id": {
          "type": "string" }}},
    {
      "type": "object",
      "properties": {
        "name": {
          "type": "string" }}},
    {
      "type": "object",
      "properties": {
        "email": {
          "type": "string" }}}
  ],
  "required": [ "id" ]
}
```

Figure 4. Examples of Equivalent Schemas Objects

D. *Comparison Outcomes: Possible Compatibility*

Once the Canonical Forms of the source and destination schemas are obtained, the schemas are then compared to detect a “compatibility” or a “possible compatibility” (Table VI). This outcome depends on the usage of composition operators.

If a response schema S_{res} uses the operators: `oneOf` or `anyOf` or both at the same time, this means that the target schema should be compared to a set of possible response schemas. If a compatibility is detected, it is considered a “possible compatibility” because the match is not guaranteed. Instead, if the request schema uses the `oneOf` or `anyOf` operators, this means that the sink API accepts data conforming to different schemas. In this case, if at least one of the request schemas matches (one of) the response schema we classify it as “compatibility” following Postel’s law. This also holds if both request and response schemas include exactly the same set of schemas inside the `oneOf` or `anyOf` operator.

In all the other cases, if the canonical forms are identical, then the schemas are considered “compatible”.

IV. Results

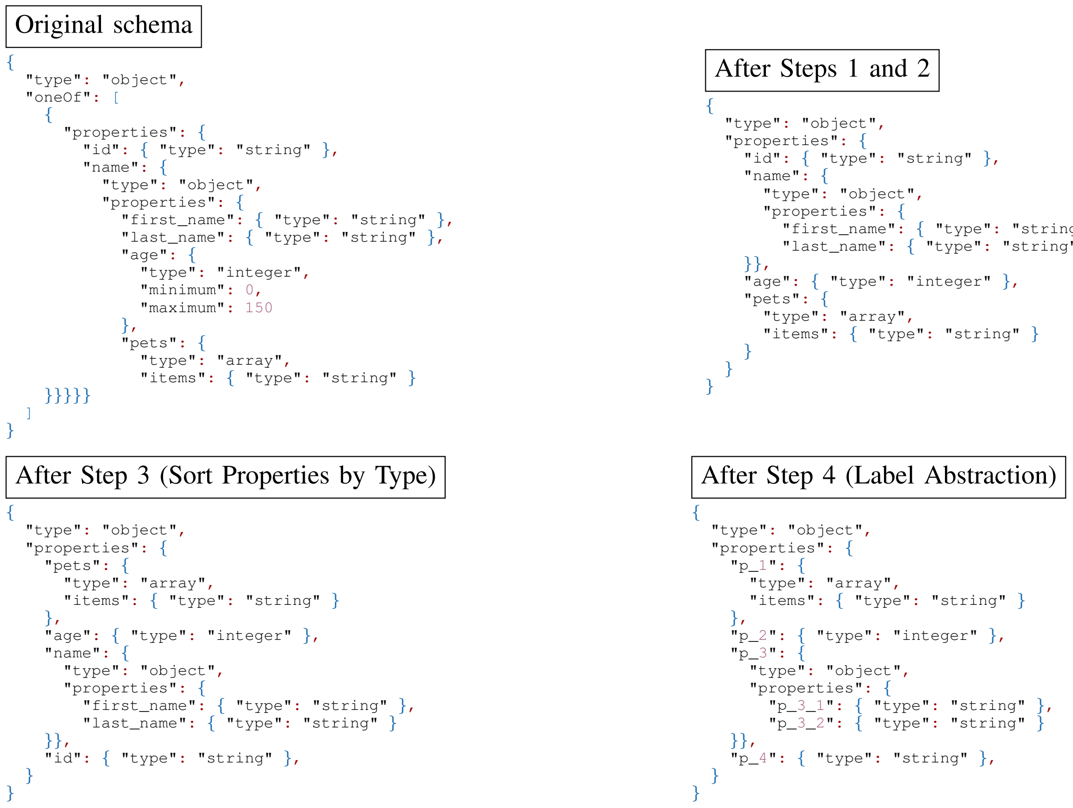
Q1: *How compatible are Web APIs?*
Overall, we found 358,136,317 pairs of endpoints belonging to 25,461,429 pairs of source and sink APIs that can potentially be directly composed. Table IV shows how the number of matches depends on the considered matching level. As expected, less matches are found when considering both data types and property names.
In the case of Property-Name Level matches the composability depends on the sink schema data types. Among the 4,488,045 detected Property-Name Level API matches we found 3,012,623 cases where the endpoints are Directly Composable: 511,563 because they also match also on a Data-Type Level, while for 2,501,060 it is possible to perform automatic data conversion of all properties. In 683,929 matches we found schemas with identical properties name, but different data types. However, in these cases the properties of the sink schemas are all of `string` type, also making the data adaptation trivial, thus we still consider the endpoints to be potentially directly composable. While, in the case of 791,493 matches, the data types conversion of all the properties is not evident, making those more challenging to indirectly compose.

Q2: *On which level are most API compatible?* The results shown in Table V indicate that there are less APIs (54.6% of sources and 31.9% of sinks) which use the same property names than the ones (92.6% sources, 43.6% sinks) which share the same property types. When considering both criteria, the number of compatible APIs drops to 19.4% (sources) and 18.5% (sinks).

Q3: *What is the likelihood that a random pair of APIs has at least one compatible endpoint?* Based on the endpoints compatibility results (Table VI), we compute the likelihood that an APIs with a specific number of endpoints n has at least one endpoint compatible with any other endpoint of APIs present in the collection. We define this probability as:

$$p_L(n) = \frac{|A_n^{comp_L}|}{|A_n|} \tag{1}$$

where $A_n^{comp_L}$ is the set of APIs with n endpoints and with at least one compatible endpoint according to a specific matching level L. A_n is the full set of APIs having exactly n endpoints. In Fig. 7 we plot the probabilities $p_{S\&S}(n)$ of matches occurring on the Data-Type and Property-Name Level for APIs with endpoints $n \leq 500$.
We can see that the more endpoints an API has, the more likely it is to have at least one compatible endpoint. The probability drops significantly for the APIs with $n \leq 100$. Only for 130



```
{
  "type": "object",
  "oneOf": [
    {
      "properties": {
        "id": { "type": "string" },
        "name": {
          "type": "object",
          "properties": {
            "first_name": { "type": "string" },
            "last_name": { "type": "string" },
            "age": {
              "type": "integer",
              "minimum": 0,
              "maximum": 150
            },
            "pets": {
              "type": "array",
              "items": { "type": "string" }
    }}}}}
  ]
}
```

```
{
  "type": "object",
  "properties": {
    "id": { "type": "string" },
    "name": {
      "type": "object",
      "properties": {
        "first_name": { "type": "string" },
        "last_name": { "type": "string" }
    }},
    "age": { "type": "integer" },
    "pets": {
      "type": "array",
      "items": { "type": "string" }
    }
  }
}
```

```
{
  "type": "object",
  "properties": {
    "pets": {
      "type": "array",
      "items": { "type": "string" }
    },
    "age": { "type": "integer" },
    "name": {
      "type": "object",
      "properties": {
        "first_name": { "type": "string" },
        "last_name": { "type": "string" }
    }},
    "id": { "type": "string" },
  }
}
```

```
{
  "type": "object",
  "properties": {
    "p_1": {
      "type": "array",
      "items": { "type": "string" }
    },
    "p_2": { "type": "integer" },
    "p_3": {
      "type": "object",
      "properties": {
        "p_3_1": { "type": "string" },
        "p_3_2": { "type": "string" }
    }},
    "p_4": { "type": "string" },
  }
}
```

Figure 5. Canonical Form Transformation Example

Table IV
Number of API and Endpoint Matches where a pair of compatible request and response schemas has been found according to each level of compatibility.

Matching Level	Type	#API Matches	(Possible)	#Endpoint Matches
Property-Name	Complex	4,488,045	162,001	21,542,616
Data-Type	Complex	15,457,565	304,372	79,412,732
	Primitive	9,490,301	59,844	277,248,326
Data-Type & Property-Name	Complex	511,563	40,925	1,475,259

APIs with $n > 100$ it is not possible to compose them with any other API of the collection, while 15,516 APIs with $n \leq 100$ do not have endpoints that can be directly composed without adaptation.

The weighted average probability computed over the entire collection is 0.21.

Q4: *To what extent are the APIs endpoints compatible?* To assess this we introduce the *Compatibility Rate*: the total number of compatible endpoints over the total number of endpoints in the API. A compatibility rate of 100% on all the Matching Levels indicates that a match was found for all endpoints which have schemas that are statically comparable.

More in detail, syntactically compatible APIs have on average 2.31 sink endpoints and 2.33 compatible source endpoints, which represent in average 37% (sink) and 72% (source) of the total number of endpoints. When considering the Property-Name Level or the Data-Type Level the average number of compatible endpoints drops to 1 (while the median is higher at 3). The largest API which is fully compatible (100% compatibility rate) is a source API with also the largest number of compatible endpoints (23).

V. Discussion

Q1: *How directly composable are Web APIs?* Out of all possible pairs of APIs, we found 511,563 pairs (0.12%) which correspond to a Data-Type and Property-Name Level compatible match. It remains to be seen how many of these can be composed in a meaningful way, but at least a Mashup developer evaluating such recommendations will not have to worry about whether the JSON payloads can be successfully transferred. When relaxing the matching level, the proportion

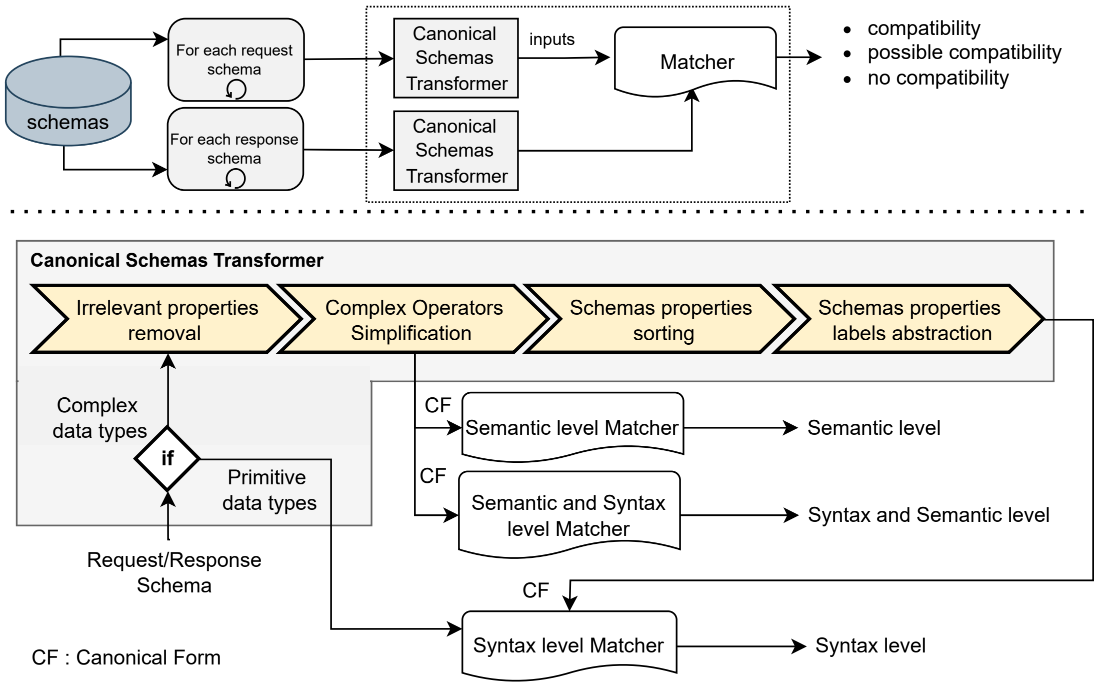


Figure 6. Schemas matching levels and canonical form transformation overview

Table V
CLASSIFICATION OF COMPATIBLE APIS AND ENDPOINTS

	Matching Level	Type	#Source	#Sink	#(Sink ∩ Source)	#(Sink ∪ Source)
APIs	Property-Name	Complex	11,236	6,566	6,171	11,631
	Data-Type	Complex	19,062	8,974	8,084	19,952
		Primitive	7,086	1,764	1,706	7,144
	Data-Type & Property-Name	Complex	4,010	3,821	3,693	4,132
	Total Number of APIs in the collection					20,587
Endpoints	Property-Name	Complex	11,413	6,601	6,234	11,780
	Data-Type	Complex	34,032	12,312	11,906	34,438
		Primitive	12,306	8,453	4,960	15,799
	Data-Type & Property-Name	Complex	4,038	3,901	3,778	4,161
	Total Number of Endpoints in the collection					364,604

of matches increases slightly (0.71% for Property-Name Level but still directly composable and 5.89% for Data-Type Level with the need to introduce property name mappings).

Q2: *Are most APIs compatible on a syntactic or Property-Name Level?* We found that a large majority of the APIs (92.6%) is syntactically compatible, where 4,132 of them are also Semantically compatible. Instead, only 56.6% of the APIs have at least one endpoint that is only semantically compatible. This implies that primitive data type conversions are likely to be less frequently required than property name mappings to compose APIs with only Property-Name Level or Data-Type Level matches.

Q3: *What is the likelihood that a random pair of APIs has at least one compatible endpoint?* On average, we found that approx. 21% of APIs have at least one compatible endpoint according to the most strict compatibility level (Data-Type & Property-Name). We did not have any expectations about the result, which is remarkable considering that APIs are published independently on the Web and do not always comply with data modeling standards. By analyzing the sample of Web APIs collected in this study, we also confirmed our expectation that such probability increases with the size of the API. Further study is needed to determine whether this result is sensitive to the size of the sample of APIs considered. As the number of APIs found on the Web keeps increasing, does the probability that they are compatible remains constant?

Table VI
NUMBER OF COMPATIBLE ENDPOINTS AND COMPATIBILITY RATES ACROSS ALL APIS

Metric	Role	Matching Level	**Min**	**Avg**	**Max**	**StdDev**	**Median**	**Q1**	**Q3**
Compatible Endpoints	Sink	Property-Name	1	1.00	4	2.25	3	1	4
		Data-Type	1	2.31	5	3.28	3	1	5
		Data-Type & Property-Name	1	1.02	4	1.62	2	1	2
	Source	Property-Name	1	1.02	23	3.69	3	1	5
		Data-Type	1	2.33	23	3.38	5	2	8
		Data-Type & Property-Name	1	1.00	23	4.91	3	1	5
Compatibility Rate	Sink	Property-Name	0.12	0.28	1.00	0.51	0.24	0.19	0.76
		Data-Type	0.31	0.37	1.00	0.40	0.58	0.35	0.87
		Data-Type & Property-Name	0.11	0.18	1.00	0.55	0.11	0.11	0.33
	Source	Property-Name	0.24	0.39	1.00	0.37	0.33	0.24	0.66
		Data-Type	0.38	0.72	1.00	0.28	0.87	0.51	0.91
		Data-Type & Property-Name	0.22	0.28	1.00	0.52	0.33	0.22	0.54

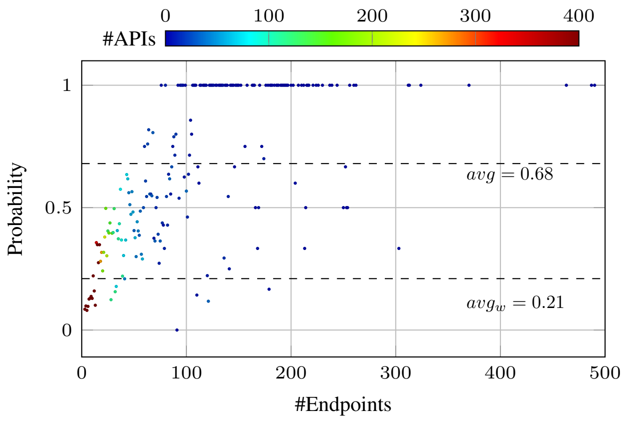


Figure 7. Probability that a random pair of APIs of a given size has at least one matching endpoint

Q4: *To what extent are the APIs endpoints compatible?* The number of Compatible Endpoints in an API is rather low, depending on the Matching Level. It goes from an average of two endpoints per API in the case of syntactic compatibility, to an average of one endpoint per API in the case of both semantic and syntactic compatibility or only semantic compatibility. The compatibility rate is also higher in the case of Data-Type Level matching where the average Compatibility Rate is of 37% (sink) and 72% (source) against lower rates in the case of Property-Name Level: 28% (sink) and 39% (source) and Data-Type & Property-Name Level: 18% (sink) and 28% (source).

VI. THREATS TO VALIDITY

External Validity. OAS descriptions are often manually created and edited, which makes them prone to errors and inconsistencies. In our data selection phase all invalid specifications were eliminated from the study. The specific sample of API descriptions analyzed may limit the external validity of the results, especially regarding the probability of finding a random pair of compatible APIs.

Internal Validity. Higher-level semantics descriptions are not generally found in OAS, making it impossible to take those into account in the compatibility definition at the core of our study. In some data models, a format qualifier is found that could be used in addition to the type information. However, the wide variety of format values found in the collection made it impractical to rely on this additional metadata.

In JSON Schema, by default any additional properties are allowed unless the `additionalProperties` field is explicitly set to `false`. However, we noticed a low usage of this construct in the extracted set of schemas only 21%, 143,341 out of 679,256 schemas of complex data type use the field). The matches found in this study did not consider the presence of such additional properties, which may overestimate the results.

VII. RELATED WORK

This empirical study is inspired by [12]: an approach to determine how two APIs can be directly composed based on analyzing the schemas of the JSON data they produce or consume at the syntactic level only. In our work, we also consider the Property-Name Level and apply the matching rules systematically to a large collection of real-world APIs.

One of the empirical studies most related to our work is [22], where the authors study a set of 8,399 GraphQL APIs with schemas written in Schemas Description Language (SDL). They found a lack of conformity to naming conversions and low adoption of pagination but did not systematically investigate schema compatibility. While in this work, we search for equivalent response/request schemas, in [?] we performed an empirical study to detect equivalent structural HTTP APIs fragments. In [17], the authors studied the relationship between the APIs structures and their data models, starting from a large set of APIs specifications (42,194 OAS).

To our knowledge, there exist no tools to compare JSON schemas considering property names and data types separately. Existing tools such as [1] consider both, without handling the challenges we identified related to alternative JSON structures to represent the same data types. Using the JSON Schema compare tool, the schemas ❶, ❷, ❸, and ❹ are considered distinct. Thanks to the canonical transformation, our comparison strategy can also detect possible matches with the `oneOf` or `anyOf` operators where the destination schema is a subset of the source schema. A similar approach, also based on canonical transformations of schemas, is used in [10] by a tool that detects data compatibility bugs based on JSON subschemas checking.

In this study, we do not assume that if two schema attributes have the same name then they are semantically equivalent. However, if their types are also similar, or if the sink attribute is of string data-type, then they are safely composable, but still not evident that such pair of endpoints can be involved in a meaningful API composition. Having additional semantic knowledge about the API data model [5] would facilitate the automatic determination of whether the endpoints with compatible schemas can also be composed. This is the goal of the authors of JWASA [21], a tool that helps developers to semi-automatically embed semantic knowledge to an API description written in natural language. Based on the description, the tool generate a JSON document to annotate the API element using JSON-LD [19], which can be later seamlessly translated to RDF, thus, allows over-passing semantic interoperability obstacles with the world of linked data. JWASA facilitates the task of generating a formal API semantic artifact, however, deciding the vocabulary to use for the annotations remains a pure manual task, and requires business domain knowledge to accurately map schema elements to ontology languages [9]. To overcome the need for this manual task, noticeable emerging Mashups recommendation tools [6], [11], [15], [23] proposed by researchers, employ Machine Learning and Deep Learning models exploiting datasets of known Mashups.

VIII. Conclusion

In this paper, we present the results of a study to assess the compatibility of a large collection of Web APIs. The work makes use of a novel large-scale schema compatibility checking tool for statically checking whether two OAS schema definitions are compatible by matching property names and/or data types. We studied the potential composability of a large collection of 20,587 Web APIs through their OpenAPI descriptions, gathered from public sources. This was performed by matching the schemas of produced or consumed data by each distinct pair of endpoints. The matching was done considering schema property names and data types both separately and together. We found that a large majority of the APIs (92.6%) have at least one compatible endpoint of which the schemas match at least on the data-type level, and the average number of this compatible endpoints is equal to to 2 endpoints per API. We also looked for stricter schemas compatibility by matching them on both property names and types and found that, on average, an API has a 21% probability that at least one of its endpoints can be composed with no need of introducing data conversion adapters as it provides (or consumes) data that is compatible with the data that is consumed (or provided) by another endpoint found in the same or different APIs.

References

[1] Json schema compare. https://github.com/mokkabonna/json-schema-compare, 2017.

[2] JSON Schema. https://json-schema.org/, 2022.

[3] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2003.

[4] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. An empirical study on the "usage of not" in real-world json schema documents. In *International Conference on Conceptual Modeling*, pages 102–112, 2021.

[5] Robert Battle and Edward Benson. Bridging the semantic web and web 2.0 with representational state transfer (REST). *Journal of Web Semantics*, 6(1):61–69, 2008.

[6] Junwu Chen, Ye Wang, Qiao Huang, Bo Jiang, and Pengxiang Liu. Open apis recommendation with an ensemble-based multi-feature model. *Expert Systems with Applications*, 196:116574, 2022.

[7] Florian Daniel and Maristella Matera. *Mashups*. Springer, 2014.

[8] Giusy Di Lorenzo, Hakim Hacid, Hye-young Paik, and Boualem Benatallah. Data integration in mashups. *SIGMOD Rec.*, 38(1):59–66, jun 2009.

[9] Paola Espinoza-Arias, Daniel Garijo, and Oscar Corcho. Mapping the web ontology language to the openapi specification. In *International Conference on Conceptual Modeling*, pages 117–127. Springer, 2020.

[10] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. Finding data compatibility bugs with json subschema checking. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 620–632, 2021.

[11] Deling Huang, Xialong Tong, and Haodong Yang. Web service recommendation based on graph attention network (gat-wsr). In *2022 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–5. IEEE, 2022.

[12] Javier Luis Cánovas Izquierdo and Jordi Cabot. Composing json-based web apis. In *International Conference on Web Engineering*, pages 390–399. Springer, 2014.

[13] Yong-Ju Lee and Jae-Soo Kim. Automatic web api composition for semantic data mashups. In *2012 Fourth International Conference on Computational Intelligence and Communication Networks*, pages 953–957. IEEE, 2012.

[14] Chune Li, Richong Zhang, Jinpeng Huai, and Hailong Sun. A novel approach for api recommendation in mashup development. In *2014 IEEE International Conference on Web Services*, pages 289–296. IEEE, 2014.

[15] Sixian Lian and Mingdong Tang. Api recommendation for mashup creation based on neural graph collaborative filtering. *Connection Science*, 34(1):124–138, 2022.

[16] Hye-Young Paik, Angel Lagares Lemos, Moshe Chai Barukh, Boualem Benatallah, and Aarthi Natarajan. *Web service implementation and composition techniques*, volume 256. Springer, 2017.

[17] Souhaila Serbout, Fabio Di Lauro, and Cesare Pautasso. Web apis structures and data models analysis. In *Proc. International Conference on Software Architecture (ICSA)*. IEEE, 2022.

[18] Souhaila Serbout, Cesare Pautasso, Uwe Zdun, and Olaf Zimmermann. From openapi fragments to api pattern primitives and design smells. In *26th European Conference on Pattern Languages of Programs*, pages 1–35, 2021.

[19] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. Json-ld 1.0. *W3C recommendation*, 16:41, 2014.

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

[21] Xianghui Wang, Qian Sun, and Jinlong Liang. Json-ld based web api semantic annotation considering distributed knowledge. *IEEE access*, 8:197203–197221, 2020.

[22] 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.

[23] Lina Yao, Xianzhi Wang, Quan Z Sheng, Boualem Benatallah, and Chaoran Huang. Mashup recommendation by regularizing matrix factorization with api co-invocations. *IEEE Transactions on Services Computing*, 14(2):502–515, 2018.

[24] Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel. Understanding mashup development. *IEEE Internet computing*, 12(5):44–52, 2008.