From OpenAPI Fragments to
API Pattern Primitives and Design Smells

Souhaila Serbout
Software Institute, USI
Lugano, Switzerland
souhaila.serbout@usi.ch

Cesare Pautasso
Software Institute, USI
Lugano, Switzerland
c.pautasso@ieee.org

Uwe Zdun
University of Vienna, Faculty of Computer Science,
Software Architecture Research Group
Vienna, Austria

Olaf Zimmermann
University of Applied Sciences of Eastern Switzerland
Rapperswil, Switzerland

ABSTRACT

In the past few years, the OpenAPI Specification (OAS) has emerged as a standard description language for accurately modeling Web APIs. Today, thousands of OpenAPI descriptions can be found by mining open source repositories. In this paper, we attempt to exploit these artifacts to extract commonly occurring building blocks used in Web API structures, in order to assist Web API designers in their modelling task. Our work is based on a fragmentation mechanism, that starts from OpenAPI descriptions of Web APIs to extract their structures, then fragment these structures into smaller blocks. This approach enabled us to extract a large dataset of reoccurring fragments from a collection of 6619 API specifications. Such fragments have been found multiple times in the same or across different APIs. We have classified the most reoccurring fragments into four pattern primitives used to expose in the API access to collections of items. We distinguish for each primitive variants from design smells. This classification is based on the specific combinations of operations associated with the collection items and on an in-depth analysis of their natural language labels and descriptions. The resulting pattern primitives are intended to support designers who would like to introduce one or more collections for a specific class of items in their HTTP-based API.

CCS CONCEPTS

• **Software and its engineering** → **Patterns**; *Designing software.*

KEYWORDS

Application Programming Interfaces, API Patterns, API Design Smells, API Structural Fragments, Collections, Pattern Minning, OpenAPI

1 INTRODUCTION

Application Programming Interfaces (APIs) open up software architectures so that the resulting software system can be integrated with external systems, developed at different times by different parties. In this paper, out of many existing kinds of APIs, we focus on Web APIs [3, 14, 17, 18, 21, 24] remotely accessible through the HTTP protocol and described using the standard OpenAPI specification language [22]. We do so because of the large number of API descriptions using this language which can be retrieved by crawling open source repositories (Figure 3). While the original purpose of OpenAPI was to generate human-readable documentation, it can also be used to generate interactive test clients, as well as client-side and server-side stubs [1, 2, 11].

In this paper, we statically analyze a large collection of real-world API descriptions looking for recurring structures that can play the roles of pattern primitives [25] which can be composed to obtain API design patterns [28]. In particular, we are interested in the resources exposed by the HTTP API naming and in the relationship between resource paths and the corresponding HTTP methods. This information can be used by clients to invoke the corresponding operations.

As shown in Figure 1, we started by crawling open-source code repositories for API description documents that use the OpenAPI specification. These documents are parsed and fed to a model from which API structure trees can be extracted. These trees are then fragmented, and the resulting fragments are matched to detect reoccurring ones. Finally, they are clustered to obtain known uses.

Our contributions include:

(1) A method to detect similar reoccurring API structures, which takes into account natural language labels associated with each path segment. This method can be also used to compare the structure of different Web APIs.

(2) A collection of widely used API fragments, with a quantitative analysis about how frequently they occur across the same or different real-world APIs.

(3) A collection of structural pattern primitives which have been used as building blocks for HTTP-based APIs. In particular we selected API structures used to provide API clients with access
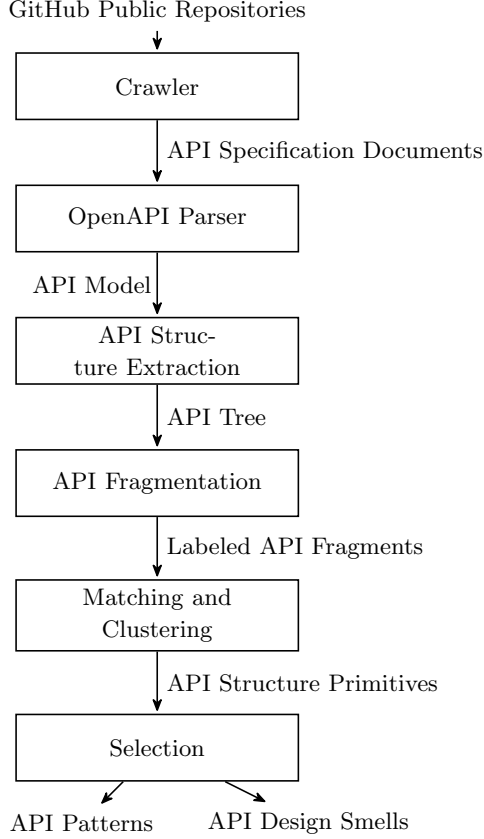
GitHub Public Repositories

```
Crawler
```

API Specification Documents

```
OpenAPI Parser
```

API Model

```
API Struc-
ture Extraction
```

API Tree

```
API Fragmentation
```

Labeled API Fragments

```
Matching and
Clustering
```

API Structure Primitives

```
Selection
```

API Patterns   API Design Smells

Figure 1: API Analytics Pipeline: From API Specifications to Patterns

to resource collections of related items (e.g., user accounts, purchase orders and their items, computational jobs, blog posts and their comments, videos or audio tracks).

(4) A classification of some design smells found across Web APIs.
(5) Two composition operators for assembling the pattern primitives into larger API structures and a proposal for connecting them with API design patterns and service contracts.

The remainder of this paper is structured as follows: In Section 2, we first give an overview about the data set of OpenAPI documents under study. Then we present our approach that consists of representing APIs as trees based on their textual documentation. We finally detail the fragmentation approach we followed in order to extract common frequent structures that can be found across same or different APIs. Section 3 presents our two-step clustering approach, which takes into accounts both the structure and the semantic closeness between the *labels sequences* attached to the nodes of a specific fragment. The collection of structural API pattern primitives is presented in Section 4. Section 5 provides two examples of how these pattern primitives can be combined to form larger API structures. It also demonstrates how the pattern primitives and fragments can be mapped to architectural patterns and interface description languages. Section 6 and Section 7 cover related work and present the

threats to validity of our work. Finally in Section 8, we present our conclusions.

2 FRAGMENTING APIS

2.1 API Collection Overview

We analyze technical API descriptions written in OpenAPI, a commonly used Interface Description Language (IDL) to specify the functional characteristics of HTTP-based and RESTful APIs, as well as selected non-functional ones (for instance, some security policies).

By mining public repositories shared on GitHub during December 2020 and January 2021, we collected a data set of 6919 API description documents, with an average size of 22.8KB, including specifications of some well-known APIs such as Twilio, Slack, Flickr APIs, Google APIs, and Amazon APIs. All the APIs described in the descriptions under study contain at least one method and one path. Because of the lack of space, in this paper, we only include one visualization of one of the largest API in the dataset (Figure 9), and other examples of smaller APIs (Appendix C) to show how the detected primitives are used as building blocks to construct the whole API's structure.

The yearly distribution of the age of the OpenAPI documents in our dataset is depicted in Figure 3. The horizontal axis refers to the year of the last commit that updated the document.

2.1.1 OAS versions distribution. The collection studied in this work contains 6619 OAS descriptions. 28.9% are written in OpenAPI 3.0 and 71.1% are written in Swagger 2.0, coming from more than 600 different providers.

The two versions are slightly different from each other on the content level, but they both allow describing API structures with almost the same level of granularity. In Figure 4, we describe the main differences between the two versions. The numbers (1), (2),.., (9), show the mappings between the sections of a description written in OAS 2.0 and their corespondents in a description document written in OAS 3.0. The first difference is in the servers details section. While in OAS 2.0 it was possible to include only one endpoint for an API, in OAS 3.0 it is allowed to include multiple server objects. Other structural rearrangements have been done in OAS 3.0 in order to increase the reusability of definitions, such as the inclusion of the Components section, where securityDefinitions, schema definitions, parameters, and responses are defined. In addition, in OAS 3.0 a Component object can also contain callback descriptions, which makes this version more efficient in describing asynchronous APIs. Moreover, OAS 3.0 improved the description of the parameters and supports more security schemes and bearer formats than OAS 2.0.

In our case, these differences between the two versions do not impact the results of the structural analysis and the APIs fragments extraction, because our study focused on the paths and methods provided by APIs, which are described in both versions.

2.1.2 HTTP Methods usage. In Figure 2 we show an overview of HTTP methods usage in a subset of the APIs under study. We classify APIs based on which HTTP methods they use following the RESTful maturity model [7], which distinguishes L0) APIs that use only one endpoint and one method from L1) APIs that use multiple endpoints and still one method associated to each endpoint, and L2) APIs which use multiple methods with multiple endpoints. Given the lack of support for describing hypermedia in OpenAPI documents,

Figure 2: API Collection Sample (sorted by Number of Paths)
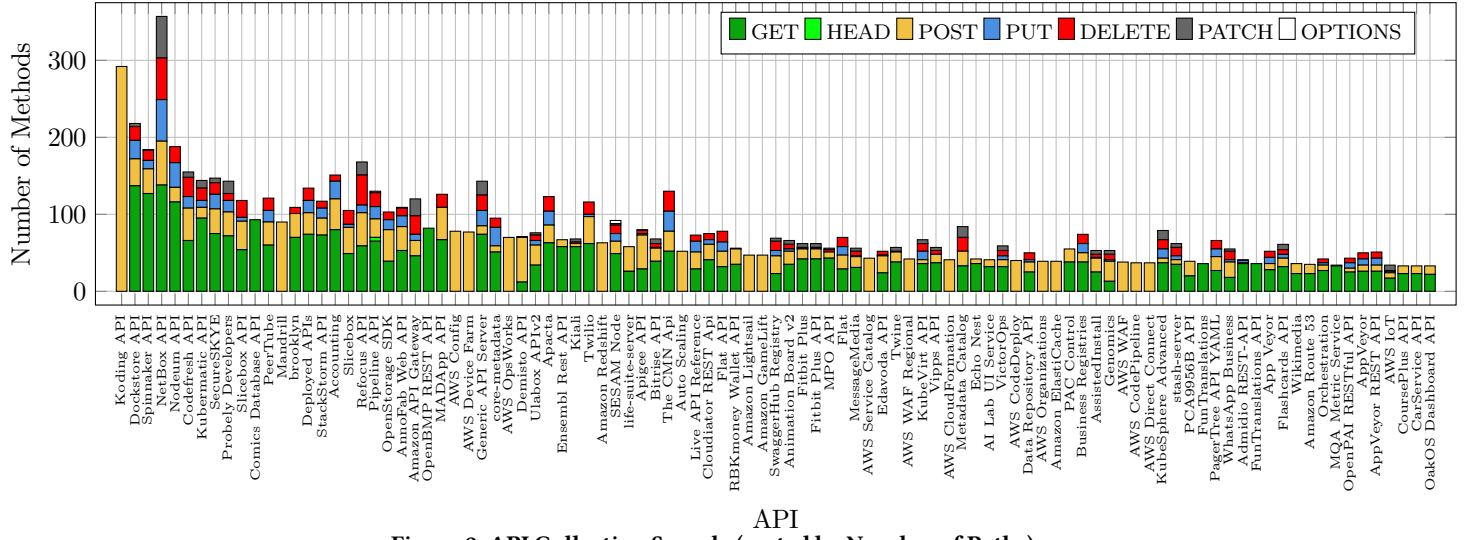
we are unable to distinguish the highest level of the maturity model L3, which includes the APIs that make use of the REST principle Hypermedia as the Engine of Application State (HATEOAS).

Still, we can clearly see different types of APIs emerging if we simply count how many HTTP methods are associated with each path enumerated in the API description (Figure 5). We have grouped the APIs into sets according to the HTTP method combinations they use and depicted the results in the bar chart in Figure 5. The most popular group makes use of the CRUD-like primitives of GET, PUT, POST, and DELETE. The second most popular group only uses the read-only GET method. This is closely followed in terms of size, by the APIs which use only the GET or POST methods. Another group of similar size can be observed by combining CRUD APIs which do not use the PUT method (so they alias update and creation operations under the same POST method) together with APIs which instead of using the PUT method they replace it with the PATCH method. The next group includes the pure RPC APIs, which only use the POST method. The last group worth mentioning is the ones that use all five methods, which includes 442 APIs. The collection also includes about 500 APIs with different method combinations, but of rather a small size.
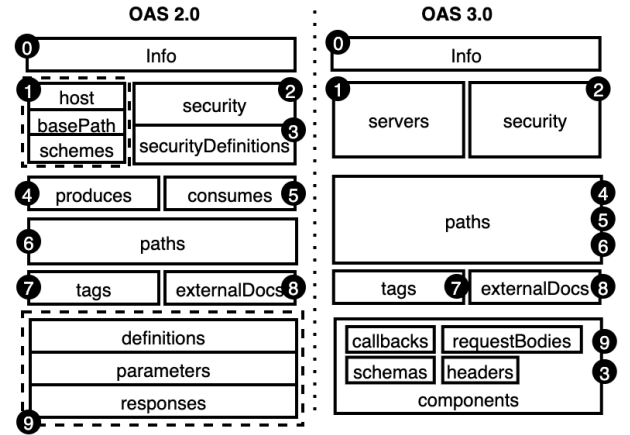

Figure 3: Yearly distribution of the age of the OAS files crawled from GitHub
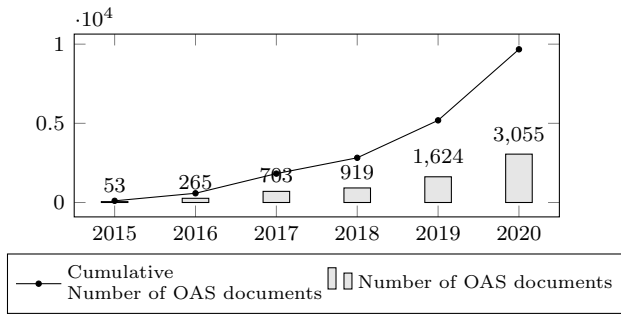
Figure 4: OpenAPI Specification Metamodel Versions: 2.0 vs 3.0

2.1.3 API sizes distribution. Figure 6 presents an overview of the size of the APIs in the same groups, measured with two different metrics [8]: a) the number of paths listed in the API description and b) the nodes present in the API tree. The boxplots in Figure 6 represent the distribution of the size measurements for each API. Overall, the median values for the size of the APIs in the collection reach approx. 50 nodes and 20 paths.

2.2 Domain Concepts

In this work, we focus on analyzing the structures of Web APIs with the goal of detecting APIs with similar structures. Due to the granularity of API descriptions in OpenAPI models, we could create a tree model representation for each API in the collection, which we call from now on API Tree. For lifting the level of abstraction of the tree model, we unlabel all its nodes. We refer to the unlabeled tree model as API Tree Structure.
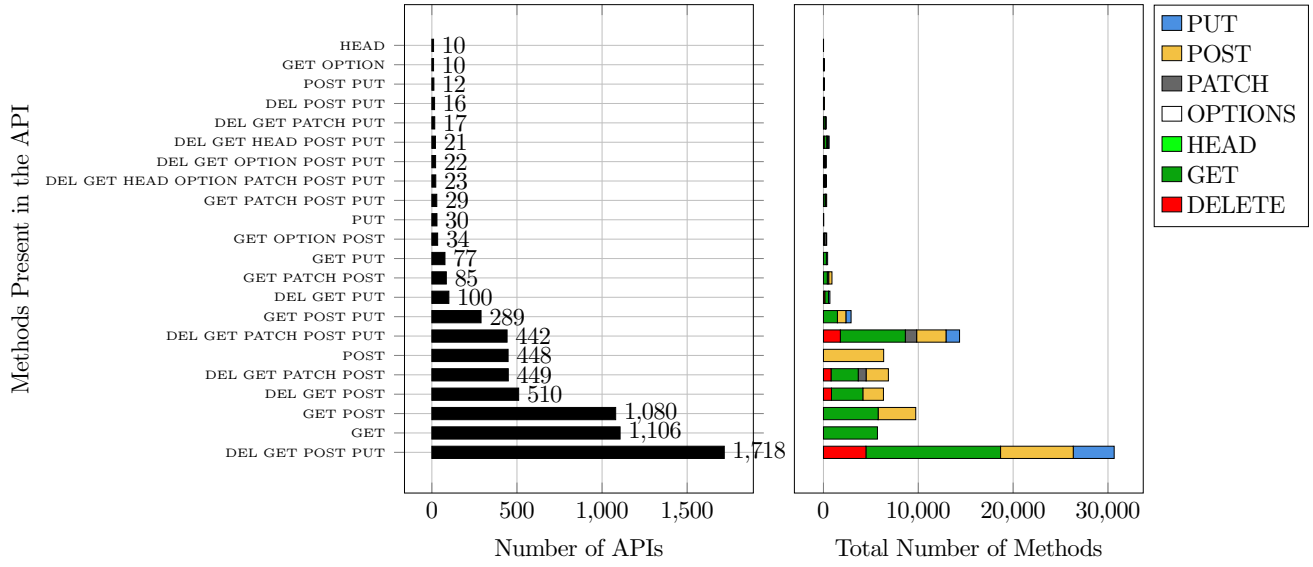
Figure 5: API Method Combination Overview
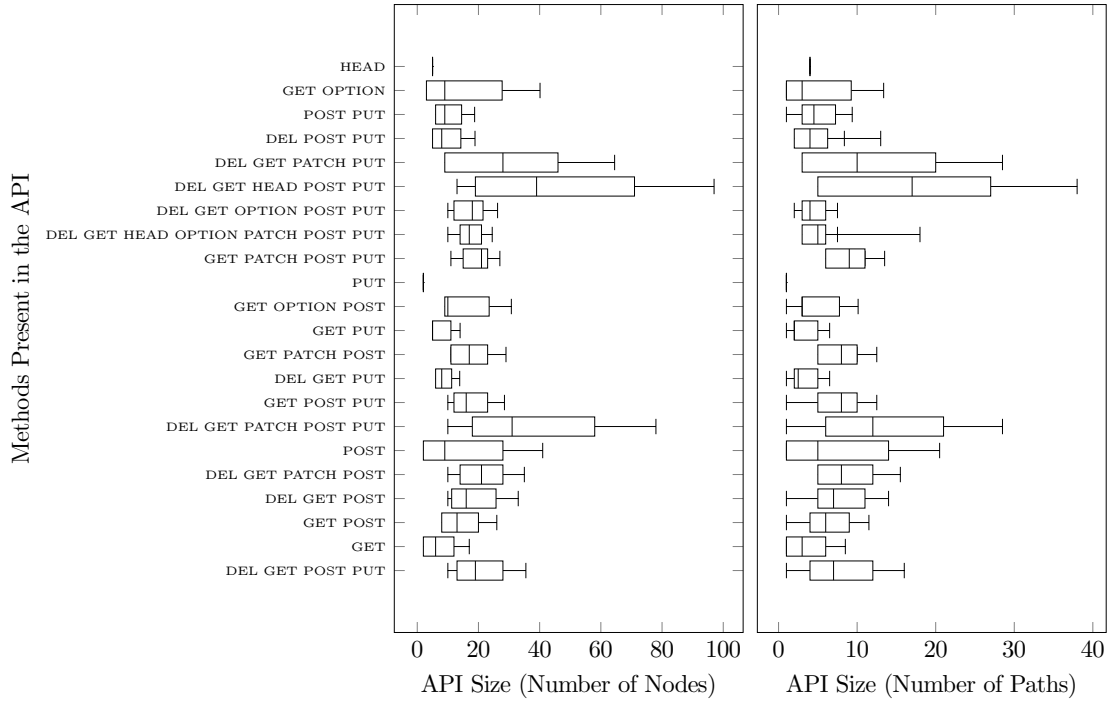
Figure 6: API Method Combination Overview vs. API Size

In our analysis, we aim to detect repetitive tree fragments in the API tree models. For that, we define an object called API Fragment, a subtree of an API tree. As for an API, the fragment also has an unlabeled version, which we call henceforth Fragment Tree Structure. After matching and filtering the set of API tree structures extracted from the whole API collection under study, we obtained a list of API structure primitives and another for API Structure smells, as described in the domain concepts summary of Figure 7.

2.3 Representing the API Structure as a Tree

2.3.1 API Tree Model. We transform the textual documentation related to the resources and the methods supported by the API into a tree data model, to represent the nesting relationships between the API endpoint URIs, enumerated as paths in the OpenAPI specification. This model has two purposes:

Figure 7: Domain Concepts and their relations

(1) It can be used to visualize the functional characteristics of the APIs graphically, to provide a quick overview supporting the understanding of the APIs structure.

(2) The second purpose of this tree data model, described in Figure 8, is to help to rapidly spot commonly used patterns by analyzing reoccurring fragments found within a large set of APIs. The elements colored in gray in Figure 8 are the ones being mapped to graphical notations for being visualized in the API Tree representation.
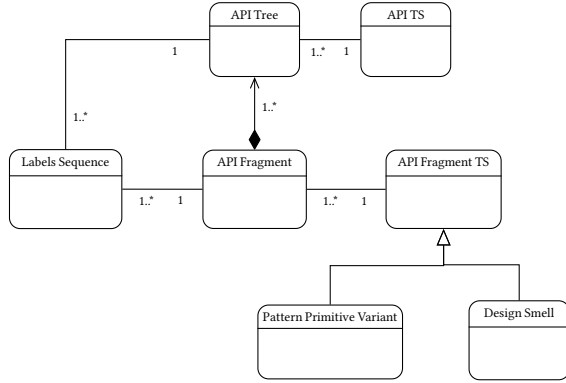
An example of an API Tree model visualization is in Figure 9. Each API operation, originally listed in the OpenAPI file, can be enumerated by following the path from the root until reaching a leaf of the tree. These later represent the HTTP methods, enumerated in each path in the Open API description. The nodes within the tree are labeled with the corresponding URI path segment and labeled depending on the type of the path segment (Table 1). The types of nodes are explained in Section 3.1.1.

Due to this graphical representation we can also visually detect a repetitive usage of an API Fragment in an API. This same fragment can also reoccur in other APIs, with different labels.

In Table 1, we summarize the notation used in our APIs Tree visualization.

Table 1: API Tree notation

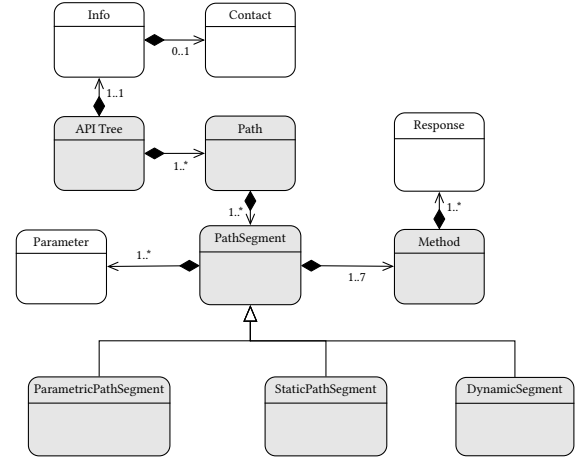| Name	Notation	Signification
Root	◯	The root of the API Tree.
Method	◯	The HTTP methods, where each method has a specific color.
Static path segment	▢	Path segment with no parameter
Parametric path segment	◼	Path segment with single {parameter}
Complex path segment	◼	Path segment mixing parameters with static labels
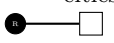
Figure 8: Excerpt of the API Tree metamodel, highlighting the visualized elements

2.3.2 OpenAPI to Tree model transformation. For explaining the model transformation, responsible of producing the tree visualization of the OpenAPI descriptions, we use the description example in Listing 1, which is an excerpt extracted from the OpenAPI description of Apacta Web API whose API Tree is shown in Figure 9.
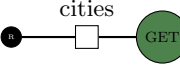
The path /cities only contains one path segment {1:cities} labeled cities. In our transformation, each segment is transformed to a PathSegment object (Figure 8). We always connect the first path segment to the Root object ●, an added graphical element which helps to visualize the API model as a tree. The {1:cities} path segment has no in-path parameters, thus it is mapped to the static path segment notation ▢ (Table 1). As a result, the obtained first portion

of the tree is ● ▢ cities , where we label the path segment node 'cities'. Moreover, the PathSegment object contains fields holding some original information such as the summary, description and the parameters information, for further usages. In this study, we only distinguish between paths that are having in-path parameters and the one that don't have them. However we plan to extend the graphical visualization to include also the other type of parameters and the responses details.

This path provides only one GET operation, which allows to get a city by its zip code. The HTTP methods are transformed to the Method object, which also keeps most of the original information about the method, such as the summary, description, and the response details. This Method object is mapped to the graphical

notation: ◯ , which contains as a label the name of HTTP method and colored in specific color depending on the method. In this case,

the notation should be 🟢GET . And as a result, the whole path visual

representation is: ● ▢ cities 🟢GET

Once all the methods of a path are all transformed, the algorithm jumps to the next path and start extracting the path segments, and put them in a list, respecting their original order. In our example the

Listing 1: Excerpt from the OpenAPI description of the Apacta API shown in Figure 9

```
paths:
    /cities:
        get:
            parameters:
                - description: Search for a city with specific zip code
                  in: query
                  name: zip_code
                  required: false
                  type: string
            responses:
                '200':
                    description: OK
                    schema:
                        ...
                '404':
                    description: Not found
                    ...
            summary: Get list of cities supported in Apacta
    /cities/{city_id}:
        get:
            parameters:
                - in: path
                  name: city_id
                  required: true
                  type: string
            responses:
                '200':
                    description: OK
                    schema:
                        properties:
                            ...
                    success:
                        default: true
                        type: boolean
                '404':
                    description: Not found
                    schema:
                        ...
```

second path is /cities/city_id. It contains two path segments { 1: cities, 2: city_id }. The path segment { 1: cities } has already been created. Knowing that each next path segment is a child of the previous one, the path segment {2: city_id} should be then connected to { 1: cities }, which is already created and added to the tree. This new node is also mapped to the PathSegment object, and more specifically to the ParametricPathSegment object, which is associated to the notation: ■. As consequence, the tree becomes :
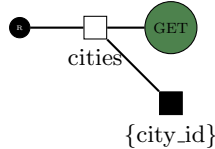
Same as for the previous path, this path also provides only one Get HTTP method. So the API Tree corresponding to the whole OpenAPI description example is:
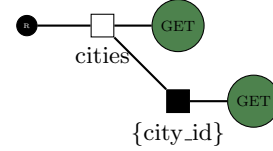
The corresponding API Tree Structure for this API Tree is simply obtained by removing all path labels:
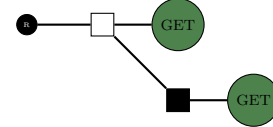
Looking at the tree model visualization in Figure 9, we can notice this same portion of the tree, constructed from the example in Listing 1, appears multiple times with different labels. For computing exactly how much frequently, a specific structure of a tree fragment appears in the set of APIs in our collection, we proceed to apply the fragmentation and matching technique presented in Section 2.4.

2.4 API Fragments

An API Fragment is any sub-tree (a connected sub-graph that includes some of the leaves of the original tree) of an API tree structure. A sub-tree is also a tree, therefore a fragment can be also seen as an API itself, which can be further decomposed. For instance, the excerpt in Listing 1, is an example of an API fragment extracted from the Apacta API (Figure 9).

To achieve our goal of detecting recurrent fragments in the API structures, we present a two-step approach that uses an algorithm that first extracts significant model fragments from a dataset of APIs models, and then compares them across multiple APIs to detect recurring ones.

2.4.1 APIs fragmentation approach. A tree T is a non-linear data structure, where each non-leaf node can be seen as a root of one or many sub-trees. The goal of the fragmentation function $\mathfrak{F} : T \rightarrow lf_1, lf_2, .., lf_n$ is to extract all the possible sub-trees $lf_1, lf_2, .., lf_n$ containing a sub-set of the ensemble of leaves of T. For collecting the nodes we walk T using Depth-First Search (DFS). In this way, we can extract all the trivial sub-trees, which are the ones having as root the different nodes of T. The algorithm extracts also non-trivial sub-trees, which are built by extracting all the branches of a sub-tree having as root a node N, then reconstructing the Tree Structures from all the possible combinations of the branches. Note that a branch starts from the root of the tree, and keeps all the methods attached to the deepest path segment node of the tree. Doing so, we obtain all the possible sub-trees having as a root the node N. Once a sub-tree is retrieved, it is serialized as JavaScript Object Notation (JSON) and stored in a MongoDB database. The same process is repeated over all the nodes of T until no node is left.

We analyse each API description in the collection and extract $T_1, T_2, .., T_m$, where m is equal to the size of our OpenAPI descriptions collection. Then we apply $\mathfrak{F}$ on each tree to extract all possible labeled sub-trees, or labeled fragments lf which include a subset of the leaves

of the tree from where it was extracted T. While labeled fragments carry the original path segments labels, unlabeled fragments f_j only distinguish whether a path segment is parametric or not, and if it contains an unusual label. The leaves of both labeled and unlabeled fragments refer to the HTTP methods which can be applied to the corresponding sub-path.

We extract all fragments from all API trees in the collection and look for reoccurring ones. To speed up the process, we first match unlabeled fragments based on their topology, then we further compare the semantic similarity of labeled fragments sharing the same structure. To do so, we project the labeled fragments lf_j into *Label Sequences* which enumerate the labels found during the traversal of each node of the API fragment tree. In other words, we apply the projection function $\mathfrak{P} : lf_j \rightarrow (TS_j, LS_j)$, to obtain for each labeled fragment a Tree Structure TS_j (also called unlabeled fragment f_j) and a Labels Sequence LS_j. All the resulted output objects are also serialized as JavaScript Object Notation (JSON) and stored in a MongoDB database.

3 FRAGMENTS CLUSTERING

3.1 API Fragments Clustering and Selection

Having obtained the set of all labeled API fragments, which in our collection corresponds to 277'094 entities, we proceed to remove duplicates and cluster them.

For clustering the fragments, we followed a two-step similarity checking approach, which consists of exact topologies matching and labels closeness similarity scoring:

(1) first by their common structure (i.e., the unlabeled fragment),
(2) then, we compute the average label semantic similarity for each cluster of fragments sharing the same structure.

The output of structural clustering consists of a set of clusters where the elements of each cluster share the same API Structure, using different labels. We give higher priority to the larger clusters (more than 40 elements), knowing that the size of the cluster reflects how common is a specific structure. These known uses are then considered as candidate structural pattern primitives. The goal behind semantically comparing the fragment sharing the same structure is to find out if there is a common use context of a highly recurring API Structure.

3.1.1 Structures matching. In our approach, we see an API fragment as a sequence of labels LS placed on the nodes of a Tree Structure TS. Where a node of a TS can be either a path segment or a leaf representing an HTTP method. During our analysis, we decided to distinguish between three types of path segment: segments containing a parameter, noted as single word label between *{ }*, segments that are not containing a parameter, and segments holding labels with more complex parameter notations, such as the example in Figure 10, which occurs 222 times. In our comparison approach, we consider the type of the path segment as part of TS. Thus, API Fragments in Figure 11 and Figure 10 are detected to be distinct since the first structural clustering step. In this way, we already distinguish fragments, which even if they have the same tree topology, have parameters in different positions along the tree.
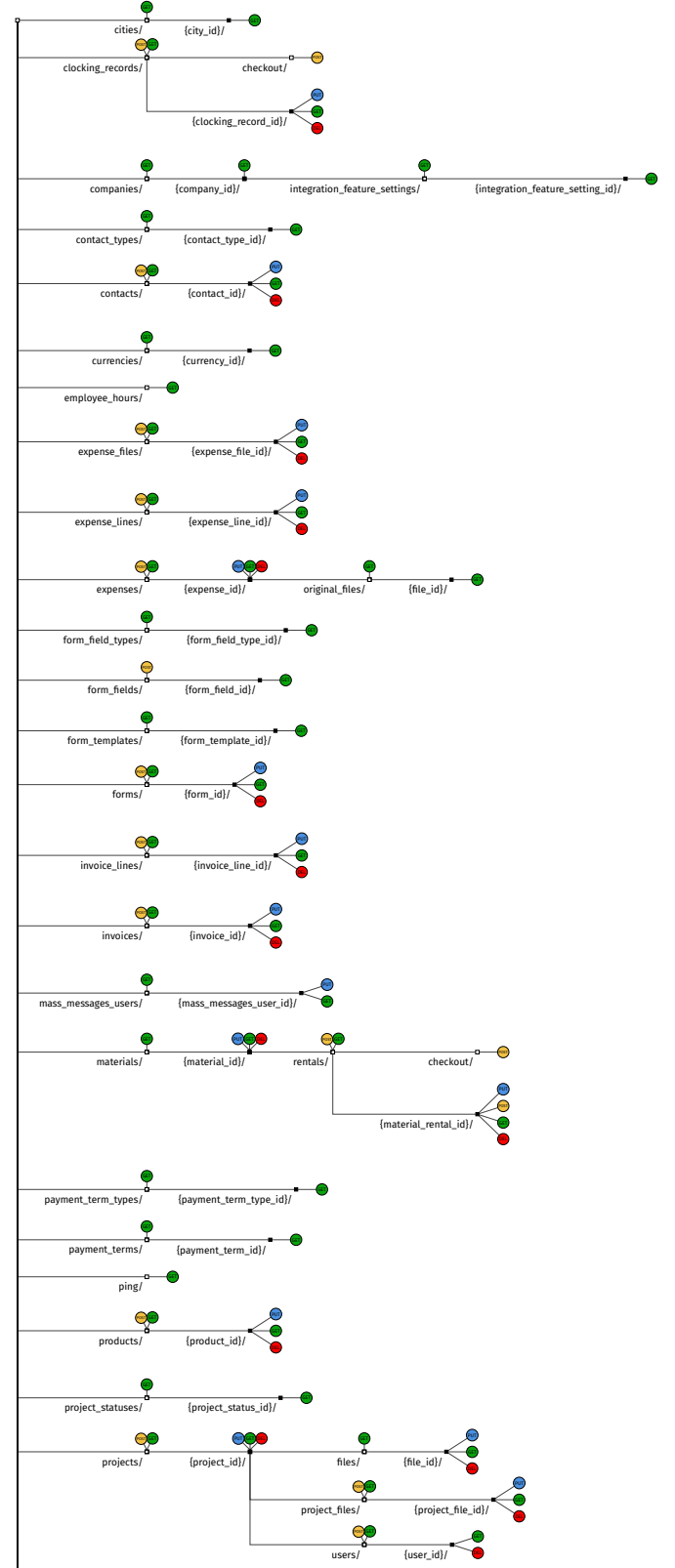

Figure 9: Visual representation of the Apacta API structure as a tree of resources and HTTP methods. This API tree includes many reoccurring subtrees, which we extract as API fragments (Click for OpenAPI source)
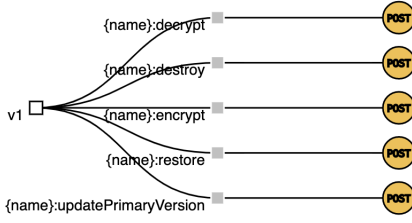
Figure 10: Example of a repetitive fragment with complex parametric path segments labels
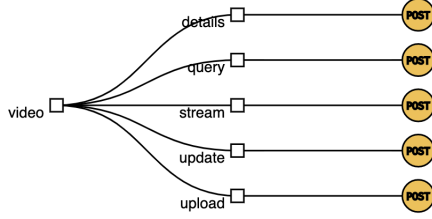

Figure 11: Example of a repetitive fragment with non parametric path segments labels

Following our fragments TS comparison approach, we extracted, from a set of 277'094 labeled fragments, 79'728 TS unlabeled fragments sharing the same tree structures, considering also the type of path segment node, and the types of the HTTP method in the leaves.

3.1.2 Semantic closeness.

Oftentimes, path segment labels carry some semantic meaning related to the resource handled by the path. For that reason, we considered taking into account the labels of the fragments nodes. Doing so, we involve the semantic context and have a better understanding of the common usage contexts of a specific fragment.

In our two-step similarity checking approach, we first clustered the fragments by their TS, then extracted all ordered sequences of node labels found for each TS of the labeled fragment (Figure 12). Doing so, we obtain a collection of labels sequences for each TS. The size of the sequence is equal to the number of nodes of the TS, excluding the leaves.

To compute the similarities between the labels sequences, we use spaCy[1], an open-source library for Natural Language Processing (NLP) in Python and Cython. In our case, we use a spaCy's trained model for English language [20], using the latest version of the "en_core_web_md" model package, multi-task CNN trained on OntoNotes, with GloVe vectors trained on Common Crawl for spaCy.

We distinguish the following types of labels:

(1) Single words (i.e., stream, details, etc as in the example fragment in Figure 11),
(2) composed labels, which concatenate single words using camel-case, or a "-" or a "_" symbol (Figure 46),
(3) unusually long, complex labels (i.e., #x-amz-target=codedeploy 20141006deleteapplication),

We added a formatter to the spaCy's processing pipeline in order to cover the different labels types cases we have. We also added a

[1] spaCy: https://spacy.io/
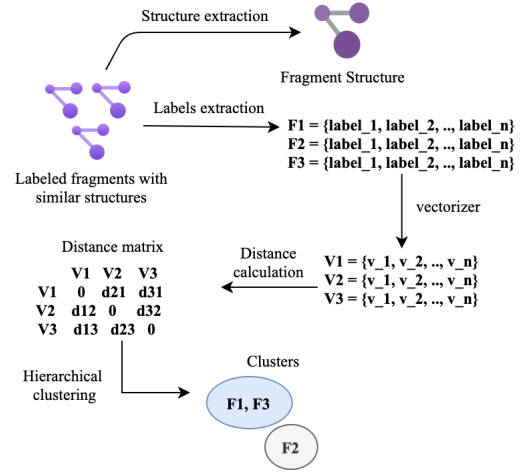
Figure 12: Fragments semantic clustering pipeline

filter at the end of the pipeline, which has a goal to exclude the labels that could not be matched to any semantic concept.

We define the distance between two label sequences $S = \{l_1, .., l_p\}$ and $S' = \{l'_1, .., l'_p\}$ as $dist(S, S') = \sum_{i=1}^{p} \frac{sim(nlp(l_i), nlp(l'_i))}{p}$. Where $sim(A, B) = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} (a_{ij} - b_{ij})^2}$ is the Euclidian Distance between the matrices $\mathbf{A} = (a_{ij})$ and $\mathbf{B} = (b_{ij})$. And where $nlp(l_i)$ is the vectorizer function of a label l_i in S. We normalize these distances to values between 0 and 10. As much is $d(S, S')$ closer to 0, S and S' are semantically close.

Doing so, within each TS cluster, we measure the semantic closeness of each sequence by calculating a similarity score between the labels attached to the same nodes of the tree. This score consists of the distance between the vectors representing the labels sequences of each fragment. Using Agglomerative Hierarchical Clustering, we obtain the semantic clusters for each set of structurally similar fragments, by setting a threshold depending on the similarity score distribution in each TS cluster.

3.2 Labels Similarity Results

While we do not have space to include the complete clustering results, we summarize the results with five metrics (Table 2):

(1) The average distance between each couple of sequences: the goal of this metric is to depict how much are each two labels sequences are alike or similar. A low average means that most of the labels sequences are composed of semantically close elements.
(2) The median of these distances: the median gives an idea about the distribution of the distances. A high median means that the majority of the labels sequences are not semantically close.
(3) The maximum distance between a couple of sequences.
(4) The number of clusters that sequences were grouped by.
(5) The threshold defining the maximum distances between all observations of two sets. This value was defines based on the distribution of the values in the distance matrix.

Primitive	Variant/Smell	Labels sequences distances			Clusters	Threshold
		Average	Median	Max		
ENUMERABLE COLLECTION P1	GET (P1v1)	3.07	5.87	9.97	218	5
	GET/PUT (P1v2)	2.60	5.23	9.01	20	5
	PUT/DEL (P1v3)	2.94	5.51	9.03	13	5
	GET/PUT/DEL (P1v4)	2.89	5.80	9.79	34	5
	GET/POST (P1s1)	3.26	7.53	9.42	14	5
	GET/DEL (P1s2)	2.54	5.56	8.65	8	6
APPENDABLE COLLECTION (P2)	GET/PUT/DEL (P2v1)	2.41	2.16	9.87	24	5
	GET/DEL (P2v2)	1.90	0.00	9.86	23	5
	GET (P2v3)	3.06	5.58	9.98	52	5
	PUT/DEL (P2s1)	2.54	4.98	8.77	24	5
	DEL (P2s2)	3.33	6.78	9.48	23	6
COLLECTION (P3)	GET/PUT/DEL/PATCH (P3v1)	1.93	3.06	9.28	19	5
	GET/PUT/DEL (P3v2)	2.62	5.02	9.81	120	5
	GET/DEL/PATCH (P3v3)	2.74	5.31	9.84	12	5
	GET (P3v4)	2.78	5.17	9.98	36	6
	PUT/DEL (P3v5)	2.54	4.72	9.91	25	5
	GET/DEL (P3v6)	3.07	5.77	9.98	39	6
	DEL (P3v7)	2.59	5.34	8.50	39	5
	PUT-ONLY (P3s1)	2.58	5.34	8.50	12	5
	GET/PUT (P3s2)	2.32	4.41	9.16	24	5
MUTABLE COLLECTION (P4)	GET/PUT/DEL/PATCH (P4v1)	1.22	0.00	8.08	19	4
	DEL (P4s1)	2.55	5.49	9.70	10	5
	GET/DEL (P4s2)	2.51	0.00	8.93	20	5

Design Smells: ☐ Create without Delete, ☐ Delete without Create, ☐ Ambiguous POST, ☐ Ambiguous PUT, ☐ Write-only

Table 2: Overview of distances between all the labels sequences of each primitive and its variants/smells. The smells are color-coded.

Table 2 shows that label sequences in different collections of fragments are semantically similar. For each primitive, we will provide detailed examples of labels associated with each variant/smell in the next section.

4 STRUCTURAL API PRIMITIVES

Out of the results obtained from the fragmentation and clustering process, we selected a set of most occurring fragments and classified them to four primitives (Figure 14), depending on their functionality based on their structures.
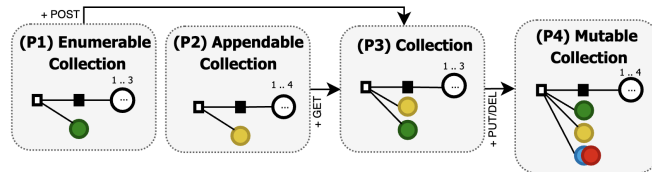
Figure 14: Overview: API Structure Collection Primitives

The *context* for all pattern primitives is the same: a designer needs to use an HTTP-based API to provide access to a collection of items which are stored on the server.

All the primitives are used to expose in the API collections of items, where each collection if identified by a statically-named container resource and its items are dynamically addressed within the container resource. We distinguish each primitive based on which combination of HTTP methods are attached to the container resource.
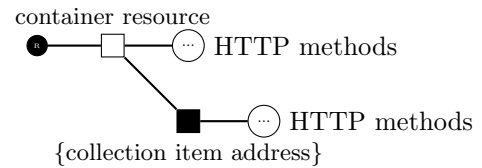
The ENUMERABLE COLLECTION (P1) primitive is used when clients can use the API to only discover the content of the collection by retrieving a list of their items. The APPENDABLE COLLECTION (P2) primitive makes it possible for clients to only append items into the collection exposed by the API. The COLLECTION (P3) primitive combines both features of the APPENDABLE COLLECTION and the ENUMERABLE COLLECTION, so that clients may use it to both append new items and list existing items. Since this primitive is the most commonly found one, we choose to name it with the simplest and shortest name, while adding qualifiers to the names of the other primitives. Finally, the MUTABLE COLLECTION (P4) primitive extends the COLLECTION with the ability to perform batch operations on the entire collection (e.g., to delete the entire content or replace the entire content of the collection).
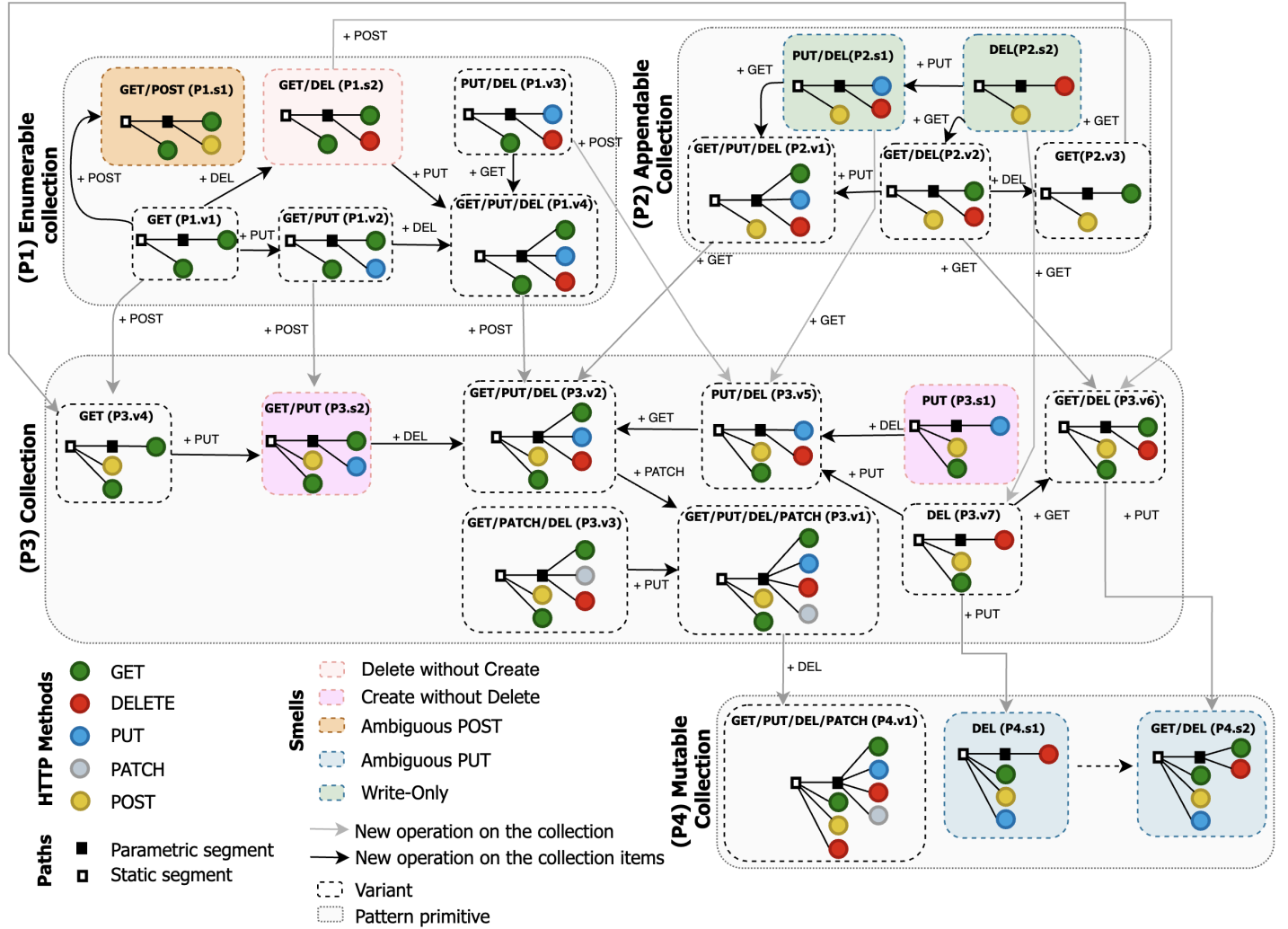
Figure 13: Overview: API Structural Primitives and their variants and design smells

Within each primitive, we have collected many variants and design smells depending on which combination of HTTP methods is attached to the collection item resource.

In Figure 13, we provide a more detailed overview showing for each primitive the corresponding variants and design smells. Each variant and design smell of the same primitive are encapsulated in a gray frame. We also show how each variant can be obtained by changing another one with the black and gray arrows. The black arrows trace the paths that allow moving from a structure primitive to another by adding an operation on the items of the collection. And the gray ones are showing which methods are added to the container resource. In the rest of this section we present overviews focused on each structural primitive.

During our analysis, we have also detected some structural design smells, which we highlight in Figure 13 with colored frames. We classified the detected smells into the following categories:

☐ Create without Delete: API structures that allow the clients to create elements from a collection, but do not provide a possibility to delete elements from it.

☐ Delete without Create: API structures that allow the clients to delete elements from a collection, but do not provide a possibility to append elements to it.

☐ Ambiguous POST: API structures that contain a POST operation on the items of a collection. Is this POST method used to append items to the collection?

☐ Ambiguous PUT: API structures that provide a PUT operation on the collection. Is this PUT method really used to update the whole collection?

☐ Write-Only: API structures that have no read operation neither on the whole collection nor on its items.

In Table 3, we show an overview of the selected collections of fragments, by listing their occurrences and the number of unique labels sequences used by the same structures in the same API or

Primitive	Variant	Occurrence	Size	nDLS	Most distant labels sequences	
					Sequence 1	Sequence 2
ENUMERABLE COLLECTION P1	GET (P1v1)	1588	4	744	$\{operations, \{operation\}\}$	$\{cidades, \{nome\}\}$
	GET/PUT (P1v2)	99	5	43	$acls, \{user\}$	$users, \{id\}$
	PUT/DEL (P1v3)	40	4	26	$\{song, \{id\}\}$	$\{oauth, \{provider\}\}$
	GET/PUT/DEL (P1v4)	176	6	71	$countries, \{country\}$	$namespaces, \{namespace\}$
	GET/POST (P1s1)	77	5	22	$msgs, \{username\}$	$servers, \{framework\}$
	GET/DEL (P1s2)	56	5	26	$\{serveurs, \{id\}\}$	$\{tasks, \{task\}\}$
APPENDABLE COLLECTION (P2)	GET/PUT/DEL (P2v1)	194	6	64	$item, \{itemid\}$	$bucketlist, \{id\}$
	GET/DEL (P2v2)	145	5	43	$vim, \{vim\_uuid\}$	$v1, \{album\}$
	GET (P2v3)	202	4	127	$\{user2, \{username\}\}$	$\{disease, \{disease\}\}$
	PUT/DEL (P2s1)	50	5	31	$rulesets, \{rulesetName\}$	$pelicula, \{peliculaId\}$
	DEL (P2s2)	69	4	51	$\{token, \{iat\}\}$	$\{jobs, \{id\}\}$
COLLECTION (P3)	GET/PUT/DEL/PATCH (P3v1)	328	8	159	$lenses, \{key\}$	$movies, \{movie\}$
	GET/PUT/DEL (P3v2)	1123	7	574	$nodes, \{ip\}$	$tickets, \{tid\}$
	GET/DEL/PATCH (P3v3)	232	5	139	$rooms, \{key\}$	$taxrate, \{zipcode\}$
	GET (P3v4)	323	5	168	$\{deposits, \{depositor\}\}$	$\{txs, \{txid\}\}$
	PUT/DEL (P3v5)	169	6	84	$pedidos, \{numero\}$	$countries, \{code\}$
	GET/DEL (P3v6)	345	6	187	$applications, \{appid\}$	$caixa, \{codigo\}$
	DEL (P3v7)	201	5	87	$byon, \{id\}$	$client, \{pubkey\}$
	PUT-ONLY (P3s1)	78	5	38	$\{Chems, \{chemid\}\}$	$\{users, \{userid\}\}$
	GET/PUT (P3s2)	63	6	47	$manager, \{username\}$	$pessoas, \{idPessoa\}$
MUTABLE COLLECTION (P4)	GET/PUT/DEL/PATCH (P4v1)	74	9	52	$workflows, \{name\}$	$boards, \{id\}$
	DEL (P4s1)	48	6	18	$progress, \{ordinal\}$	$beverages, \{beverage\}$
	GET/DEL (P4s2)	102	7	56	$ciudad, \{id\}$	$themes, \{uuid\}$

Design Smells: ☐ Create without Delete, ☐ Delete without Create, ☐ Ambiguous POST, ☐ Ambiguous PUT, ☐ Write-only

Table 3: Known Uses of Selected Fragments: Number of Distinct Label Sequences (nDLS) and their Occurrences within APIs.

across different ones. We also give an example of the most distinct sequences found among the unique labels sequences, in order to show the extreme use contexts for each structure.

The rest of this section details each of the selected primitives where we present for each primitive the different occurring variants. For each variant of each primitive we filtered the most frequent labels used by all the variants of a primitive, and sort them alphabetically to ease the readability of the heatmaps (Tables 5, 7, 9, and 12). In the Figures, we show the occurrences (counting how many times a Labels Sequence is used for the same TS) of each cluster of labels in a specific variant/smell.

We also provide a set of guidance tables, based on known uses. The listed labels are obtained by clustering the Labels Sequences by the container resource label, as explained in Appendix D.

The goal is to support designers who would like to introduce a collection for a specific class of items in their API. They can take advantage of the observations we have collected as they attempt to look up the collection label and see if there is a non-ambiguous mapping to a given primitive variant.

In order to give an idea about the yearly distribution of the variants ages and popularity, we calculate the number of APIs in which a specific variant appears (Tables 4, 8, 10, 11).

4.1 Enumerable Collection (P1)

Summary. Expose an enumerable set of items within their own container resource.

Problem. How to make the collection items discoverable by clients?

Solution. Provide a unique address for each collection item. Allow clients to read the content of each items applying the GET method to the address of the item. Group together related items under the same resource path prefix. And, allow clients to enumerate the items within the collection by applying the GET method to the container resource.
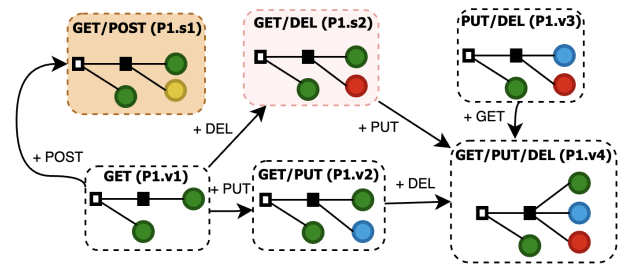

Figure 15: Enumerable Collection - Overview of Variants and Design Smells

In Table 4, we can clearly see the increasing usage of the Variants and Smells in the API collection over time. This increase can be both because of the yearly distribution of the API specifications gathered in our data set, and to the popularity the structural primitives gained through the years.

		2015	2016	2017	2018	2019	2020
	P1.s2	0	0	6	6	9	34
	P1.s1	0	1	3	7	8	24
	P1.v4	0	0	12	23	30	61
	P3.v3	0	1	5	3	5	20
	P2.v2	1	3	4	11	15	35
	P1.v1	7	17	80	86	165	374

Table 4: Yearly distribution of the API specifications where the Enumerable Collection (P1) variants appear

- **Enumerable Collection Variants**

For the Enumerable Collection primitive we have identified 2 variants and 3 design smells (Figure 15).

GET (P1.v1). The read-only variant is one of the most occurring structures, which allows clients only to enumerate the content of the collection and to read the corresponding items. APIs use it to publish one immutable set of related items. By setting a threshold of 5 obtained 218 Labels Sequences clusters. Which depicts the variety of usage contexts of this variant.

According to the whole labels sequences set that we extracted, we noticed that this read-only structure is widely used for different domains. In Table 5, we show some of the labels clusters used by this variant. We can notice that all the most frequent labels in the Enumerable Collection (P1) are used by this variant, except 3 ones: *keys* and *episodes*, which are used by the variant GET/PUT/DEL (P1.v4) which allows also to update and delete the items of the container resource, and *client*, which is only used by GET/PUT (P1.v2).

An example of API where this structure primitive is present several times, in the Apacta API showed in Figure 9. In this API we can see clearly the high occurrence of GET (P1.v1) with different labels, combined with variants of other primitives.
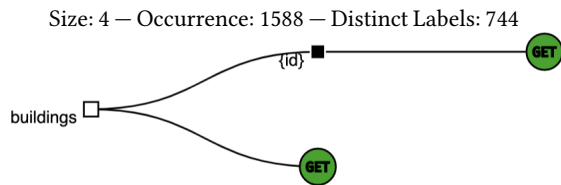
Size: 4 — Occurrence: 1588 — Distinct Labels: 744

Figure 16: Enumerable Collection - GET Variant (P1.v1) Visualization

GET/PUT (P1.v2). This variant allows clients to use the GET and PUT methods on the collection items. This makes it possible to read and update the content of individual collection items. This API structure also appears in Apacta API (Figure 9). Figure 47, is a use case example of this API structure. The GET operation in the resource handled by the path /users/id/topics allows the client to get all the topics of a specific user. The get operation in the path /users/{id}/topics/{topic_id} has as goal to verify if a user is following a specific topic. The response is an object of boolean type. In this case, the PUT operation is for interpolating the

	P1.v1	P1.v2	P1.v3	P1.v4	P1.s1	P1.s2
accounts	6	0	0	2	0	0
api-docs	9	0	0	0	0	0
applications	8	0	0	0	0	0
artifacts	12	0	0	0	0	0
clients	0	0	0	0	17	0
concepts	8	0	0	0	0	0
config_schemas	8	0	0	0	0	0
configs	2	8	0	0	0	0
content	16	0	0	0	0	0
currencies	13	0	0	0	0	0
descriptor	10	0	0	0	0	0
devices	8	0	0	8	0	0
documents	13	0	0	0	0	0
email_history	9	0	0	0	0	0
episodes	0	0	0	10	0	0
events	29	0	0	0	0	0
files	7	0	0	6	0	0
groups	9	0	0	0	0	0
health_profile	8	0	0	0	0	0
health_profile_answer	8	0	0	0	0	0
health_question_definition	8	0	0	0	0	0
history	7	0	0	0	0	1
images	9	1	0	0	0	1
instances	1	0	0	0	0	21
items	9	0	0	0	0	0
jobs	2	10	0	0	0	1
keys	0	0	0	11	0	0
locations	24	1	0	0	0	0
manifests	29	0	0	0	0	0
metadata	8	0	0	0	0	1
namespaces	4	3	0	1	0	0
networks	3	0	0	6	0	1
operations	51	0	0	0	5	0
organizations	16	0	0	0	0	0
overview	8	0	0	0	0	0
people	4	0	0	6	0	0
policydefinitions	3	0	0	6	0	0
products	13	1	0	0	0	0
resources	33	0	0	3	0	0
roles	9	0	0	0	0	0
servers	3	0	0	0	17	0
services	16	0	0	0	0	0
shows	5	0	4	6	0	0
tags	8	0	3	0	0	0
tasks	5	0	0	2	0	5
types	26	0	0	0	0	0
users	29	1	4	11	1	1
versions	9	0	0	0	0	1
views	16	0	0	0	0	0
vuln	16	0	0	0	0	0

Table 5: Enumerable Collection (P1) – Labels found in each variant/smell

FOLLOW / UNFOLLOW relationship between the user {id} and the topic {topic_id}.

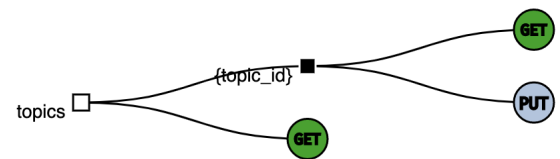Size: 5 — Occurrence:99 — Distinct Labels: 43

Figure 17: Enumerable Collection - GET/PUT Variant (P1.v2)

PUT/DEL (P1.v3). The particularity of this variant is that it allows to both update and delete items, however, it does not allow the client to create new items in the collection by using the container resource. Instead, it still allows them to do so by invoking the PUT method directly on the items to be created. In this case, clients themselves should provide the identifiers for the items to be added to the collection.
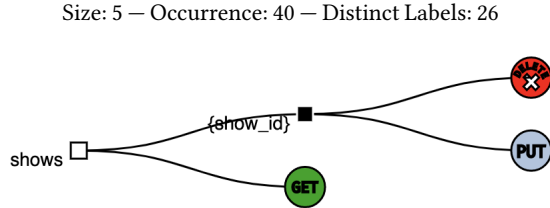
Size: 5 — Occurrence: 40 — Distinct Labels: 26

Figure 18: Enumerable Collection - PUT/DEL Variant (P1.v3)

This API structure appears one in the TVmaze user API showed in Figure 46. Where it is used for reading the collection of shows, and deleting or updating each. Another API example where this structure appears is Invotra API (Figure 45). In this case, adding a new user to the users' collection is possible due to the POST operation of the path `users`. However, it seems that the client is not allowed to add new user memberships to a specific team. While, according to the descriptions of the operations, it is possible to remove a user's membership of the team or update information about his team membership. Then, how can a team have new members? In this case, the user object schema is having a `teams` property of array type. Thus, adding a new member to a team is performed by means of the PUT operation provided in the path: `/users/{userId}`.

GET/PUT/DEL (P1.v4). The main characteristic of this structure is that, in addition to the GET and PUT methods, it also exposes a DELETE method on the collection items. This way, clients can not only read and write the associated content but can also remove items from the collection.
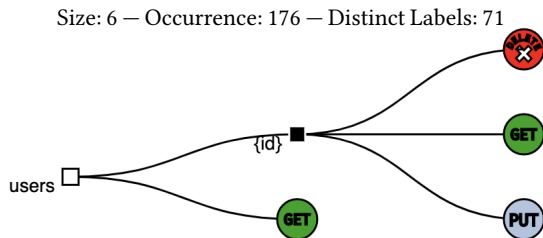
Size: 6 — Occurrence: 176 — Distinct Labels: 71

Figure 19: Enumerable Collection - GET/PUT/DEL Variant (P1.v4)

While in general, it can be useful to allow clients to remove items from a collection, it is not clear whether an API should support this for collections whose content can only be enumerated without providing the means for the service to mint identifiers for new items. Instead, new items can only be added by clients as long as they provide the new item's identifier.

While this can lead to crashes when multiple clients attempt to invoke the PUT operation on the same item, we have observed different semantics for the PUT and DELETE methods. For example, some APIs use the DELETE method for something different: task cancellation. In this case, we assume that the tasks being performed within the server can be monitored by clients and when necessary can be interrupted.

For a better understanding, we have extracted the content of the description field of the DELETE method.

Looking at the descriptions of the delete method extracted from the OpenAPI documents in Table 16 (Appendix A), it is clearly understandable that the DELETE operation is not always meant for clients to delete an item from the collection.

More in detail, in the description D-40, the DELETE method is allowing the client to delete a person from the list of followed people, but no append operation is provided. An example of an API where this fragment appears is in Figure 46. In this API example, we look at the fragment with labels sequence $S = people, \{poeple\_id\}$, in which we can notice that the following operation is done through the PUT method. In this case, when following a person, this new followed person is not appended to a collection of followed people, but instead, the followed person is updated through the PUT operation with the information about a new follower.

Path segment	Method	Description
people	GET	List the followed people
{people_id}	GET	Check if a person is followed
	DELETE	Unfollow a person
	PUT	Follow a person

Table 6: Methods description of a fragment of Enumerable Collection - GET/PUT/DEL Variant (P1.v4), extracted from the OpenAPI description of TVmaze user API

- **Enumerable Collection Design Smells**

GET/POST (P1.s1). ☐ Ambiguous POST As opposed to updating the content of individual items of the previous variants, in this variant, the API makes it possible to fetch the current state of each item with GET and invoke some arbitrary operation on each of them with POST.
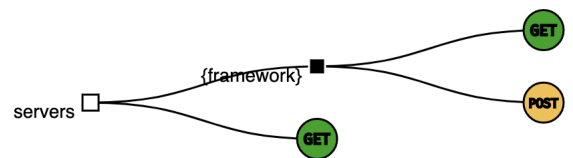
Size: 5 — Occurrence:77 — Distinct Labels: 22

Figure 20: Enumerable Collection - GET/POST Design Smell (P1.s1)

Coming back to the OpenAPI descriptions of the APIs where this variant of fragments appears, we extracted the content of the summary and description fields for the POST method, which we list in Table 15 (Appendix A). Based on the descriptions, we can detect two main use cases for the POST methods on the collection items:

(1) Appending an item to the collection: in this case placing the POST operation over the collection items can be seen as a common mistake.

(2) Updating an attribute of an existing item: in this case the POST is mistakenly used to perform the role of the PUT method.

GET/DEL (P1.s2). ☐ Delete without Create

This smell provides access to a collection whose items can be read and deleted, without offering clients the possibility to append new items.

This smell only appears with 26 distinct labels. In Figure 5, we can see that it appears 21 times out of 56 with labels represented by the label *instances*. This same label appears only once with the GET (P1.v1) variant. Other labels found in conjunction with this smell (e.g., tasks, jobs) would indicate uses for providing access to server-side resources which can only be monitored and eventually removed by clients, which do not have any control over their lifecycle.
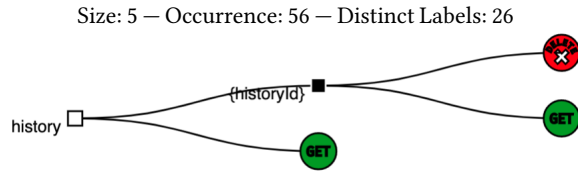
Size: 5 — Occurrence: 56 — Distinct Labels: 26

Figure 21: Enumerable Collection - GET/DEL Design Smell (P1.s2)

4.2 Appendable Collection (P2)

Summary. Append new items by posting them in the container resource

Problem. How to offer clients the ability to add new items into the collection?

Solution. Allow clients to use the POST method on the container resource to append new items into the collection. The address of the newly created items must be returned to the clients, since this pattern does not feature the ability for clients to enumerate the content of the collection.

	P2.v1	P2.v2	P2.v3	P2.s2	P2.s1
account	1	0	6	0	0
annotations	3	0	0	0	0
annotationsets	3	0	0	0	0
assets	0	0	0	3	0
batch	0	0	2	0	0
bookings	0	1	0	1	0
buy	0	0	2	0	0
campaigns	0	2	0	0	0
cart	0	1	2	0	0
categorias	0	0	0	0	4
category	1	0	0	0	1
client	1	0	1	0	0
cluster	0	2	0	0	0
collections	0	0	3	0	0
comment	0	1	0	1	5
connections	0	0	2	0	0
courses	1	4	0	0	0
datapointers	3	0	0	0	0
deployments	0	1	11	0	0
disease	0	0	5	0	0
distributions	2	0	0	0	0
documents	0	2	0	0	0
employee	0	1	1	0	1
entries	1	0	2	0	0
files	0	1	6	0	0
form_fields	0	0	6	0	0
hub	0	2	0	1	0
images	0	0	11	1	0
individuals	3	0	0	0	0
item	1	1	3	0	0
jobs	0	0	1	2	0
labels	0	0	1	5	0
media	0	4	1	0	0
messages	2	1	9	0	0
objectstores	0	0	0	0	8
order	4	78	1	1	1
policy_keys	0	0	3	0	0
post	0	0	2	1	1
productos	0	0	0	0	4
products	2	0	2	0	0
provider	2	0	0	0	1
read	0	0	5	0	0
register	11	0	0	0	0
student	1	1	1	0	0
subscriptions	1	0	0	2	0
target	0	0	5	0	0
task	1	0	1	0	1
todo	1	12	1	0	0
token	0	0	1	11	0
user	114	5	11	3	7
wall_comments	0	0	6	0	0

Table 7: Appendable Collection (P2)– Labels found in each variant/smell

- **Appendable Collection Variants**

The common point between the variants of this primitive is that they all only allow the client to append on a collection, and to perform different operations on the items. Starting from the variant that allows all of GET/PUT/DEL operations, until the one that only allows reading the items. For this primitive, we have detected a design smell, where the client is not allowed to perform any read operation, neither of the collection nor on its items.
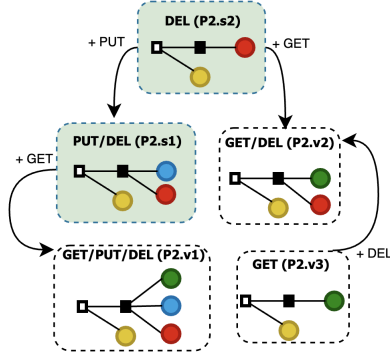
Figure 22: Appendable Collection Overview

		2015	2016	2017	2018	2019	2020
	P2.s1	0	0	0	4	12	25
	P2.s2	2	2	3	4	8	38
	P2.v3	1	3	13	20	34	104
	P2.v2	5	10	14	16	25	68
	P2.v1	4	9	19	25	53	63

Table 8: Yearly distribution of the API specifications where the Appendable Collection (P2) variants and smells appear

GET/PUT/DEL (P2.v1). This variant allows clients full control over the items they have appended to the collection, as they can read, update and delete them.

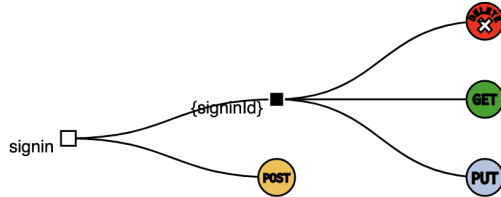Size: 6 — Occurrence: 194 — Distinct Labels: 64

Figure 23: Appendable Collection - GET/PUT/DEL Variant (P2.v1)

For understanding the reason for the absence of a read operation on the collection, we extracted natural language descriptions of the GET operation. We want to verify whether the designers mistakenly considered that the GET operation on the item would serve also for listing all the content of the collection. In Table 19, we list some of the summaries and descriptions associated with the GET method. We can see from the descriptions that the GET operation is indeed used to retrieve specific elements from the collection.

Figure 49 shows an example of use of this variant, where it is combined with the GET/DEL (P2.v2) variant. In this case, the PUT operation is used to update a sign in record.

GET/DEL (P2.v2). This variant only allows to read or delete individual collection items. It occurred 145 times, however with only 43 distinct Labels Sequences.

A concrete usage example of this primitive is in Passman API (visualized in Figure 50), an open-source developers API for Passman extensions. In the case of this example, the GET/DEL (P2.v2) variant is used in order to allow uploading and attaching a file to an item by means of the POST operation in the /file path. The client is also allowed to delete or get the content of a specific file, using, respectively the DELETE and GET operations allowed in the path /file/{file_id}. Another example is in Figure 49, where it is used beside the GET/PUT/DEL (P2.v1) variant, allowing to create a team member (user) record, to retrieve the information associated with a user's account, and finally to delete a team member's user record.

Size: 5 — Occurrence: 145 — Distinct Labels: 43
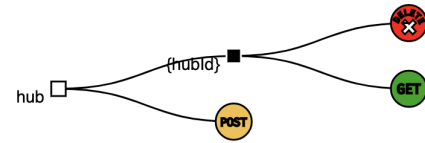
Figure 24: Appendable Collection - GET/DEL Variant (P2.v2)

GET (P2.v3). This variant only allows the client to add elements to the collection, and then read each one, but it does not provide the ability to edit or remove items. Collections featuring this primitive contain resources which are garbage collected on the server-side, such as jobs, queries, or sessions. Another example is the append-only shopping cart in which clients can only add items without ever removing them.

Size: 4 — Occurrence:202 — Distinct Labels: 127
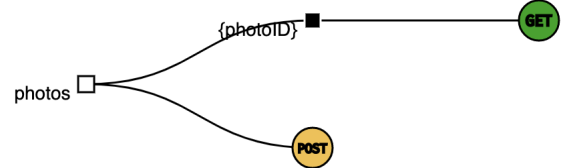
Figure 25: Appendable Collection - GET Variant (P2.v3)

- **Appendable Collection Design Smells**

PUT/DEL (P2.s1). ▢ Write-Only

Instead of a GET operation, this variant introduces a PUT. However, it does not occur as frequently as the variants GET/DEL (P2.v2) and GET/PUT/DEL (P2.v1).

We have analyzed the 50 occurrences to attempt to determine how such a write-only API fragment would work, since it appears it is only possible to append new items, update or delete them. Indeed, no occurrence supports the ability to enumerate the content of the collection, nor it allows clients to read from its items.

Size: 5 — Occurrence:50 — Distinct Labels: 31
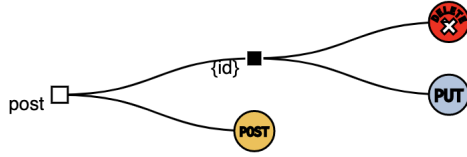
Figure 26: Appendable Collection - PUT/DEL Design Smell (P2.s1)

DEL (P2.s2). ☐ Write-Only

Same as PUT/DEL (P2.s1), this variant does not provide the client the possibility of performing GET operations. Neither on the containers nor on the items. It only allows to append new items to the collection and delete them.

Such unreadable, write-only collection can still be useful, for example, to manage asynchronous jobs, or subscriptions or messages submitted into the API which can be only canceled from the clients. Since the collection cannot be enumerated, this works only if the address of the newly created items is returned to the client who created it using POST.

Nevertheless, we tag this variant as a smell, because of the strong limitations imposed by offering a write-only collection.

Size: 4 — Occurrence:69 — Distinct Labels: 51
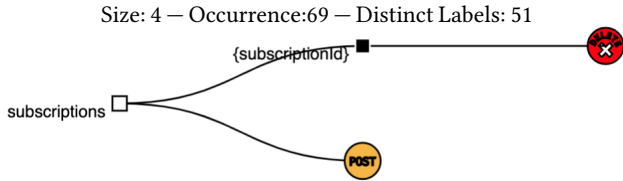
Figure 27: Appendable Collection - DEL Variant (P2.s2)

4.3 Collection (P3)

Also known as. Enumerable-Appendable Collection

Summary. Use the container resource to enumerate its content and add new items.

Problem. How to make the collection items discoverable by clients? How to let clients add items to the collection?

Solution. Group together related items under the same prefix. Allow clients to enumerate the items within the collection by applying the GET method to the container resource. Clients can use the POST method on the same container resource to add new items.

- **Collection Variants**

We present different variants featuring different method combinations on the collection item, starting from the one having four methods, all the way to fragments with a single method attached to the collection item.

In this primitive, we have detected two Design Smells (Figure 28), both are related to the ☐ Create without Delete smell.

16

	P3.v1	P3.v2	P3.v3	P3.v4	P3.v5	P3.v6	P3.v7	P3.s1	P3.s2
accounts	0	6	1	2	1	0	2	0	6
actions	0	8	3	0	0	0	0	0	0
apikeys	0	8	0	0	0	1	2	0	0
applications	2	6	0	0	0	2	0	7	0
articles	0	31	1	0	2	2	0	0	0
audit_trails	61	0	0	0	0	0	0	0	0
authorizedcertificates	0	0	13	0	0	0	0	0	0
bookings	0	3	1	0	2	0	6	0	0
books	0	9	1	0	0	2	0	0	0
categories	1	8	1	0	0	2	0	1	0
change_logs	61	0	0	0	0	0	0	0	0
cities	0	4	0	0	6	0	0	0	0
client	0	16	1	2	1	1	1	0	1
clusters	1	2	12	0	0	3	0	7	0
collaborators	0	0	0	0	0	10	0	0	0
comments	0	8	1	1	1	3	18	0	2
compositetypes	18	0	0	0	0	0	0	0	0
configs	0	2	0	0	0	0	0	0	10
contacts	0	14	0	0	0	0	0	0	1
credentials	0	0	0	0	1	0	16	0	0
domainmappings	0	0	13	0	0	0	0	0	0
events	1	16	1	0	0	1	1	0	6
example_entities	3	11	0	0	0	0	0	0	0
group	0	9	0	1	0	1	0	0	10
images	1	1	0	0	0	14	0	0	2
ingressrules	0	0	13	0	0	0	0	0	0
invoices	2	9	0	0	0	1	0	0	1
item	1	9	2	1	1	1	0	0	0
members	0	5	0	0	0	4	6	0	1
messages	0	2	4	0	0	5	4	0	6
networks	1	0	0	0	0	9	0	0	0
node	0	3	2	0	1	3	0	0	0
note	0	8	2	0	0	0	0	0	0
notifications	0	6	0	1	1	0	0	0	1
order	0	11	1	2	1	4	2	2	8
patient_health_metric	0	0	0	0	0	0	0	0	9
payments	1	5	3	0	0	0	0	0	2
pets	0	1	1	0	0	4	0	0	12
policies	0	24	1	0	0	0	0	0	0
posts	0	12	1	1	0	3	0	0	2
products	3	23	4	0	1	1	0	1	1
projects	0	24	3	1	1	8	7	8	2
reward	0	0	0	0	0	0	0	0	10
reward_earning	0	0	0	0	0	0	0	0	9
roles	1	13	2	2	1	3	0	1	3
rollouts	0	0	0	0	0	0	0	0	10
rules	0	4	1	0	8	8	1	0	8
runs	0	1	2	0	0	0	6	0	0
service-profiles	0	0	0	0	0	6	0	6	0
services	1	10	0	0	0	2	1	0	0
sessions	0	0	4	0	0	7	0	0	2
subscriptions	0	9	1	0	0	8	0	0	1
tags	1	4	3	0	26	0	12	2	1
tasks	2	3	4	1	1	2	0	0	5
tracks	0	8	0	0	1	0	1	0	1
types	8	1	1	0	1	1	0	1	1
users	4	108	10	11	31	19	7	3	30
volumes	0	0	0	0	8	7	0	0	0

Table 9: COLLECTION (P3) – Labels found in each variant/smell

Even the simplest variants with only one operation on the item to delete or update them would appear to lack the ability to directly reading individual collection items. While this is the case, as opposed to the previously discussed Appendable Collection smells, clients can still fetch the content of the entire collection using the GET operation provided by the container resource and then extract the values for individual items from the result.
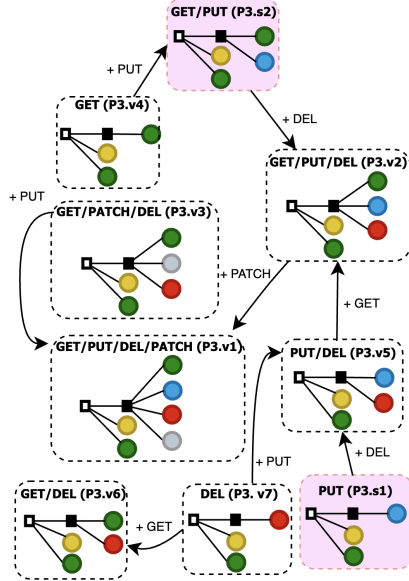
Figure 28: Collection – Overview of Variants

		2015	2016	2017	2018	2019	2020
	P3.s2	0	1	1	11	6	30
	P3.s1	0	1	6	11	14	17
	P3.v7	2	10	7	16	28	84
	P3.v6	1	3	33	43	40	134
	P3.v5	1	0	12	23	38	76
	P3.v4	1	17	33	36	51	112
	P3.v3	0	4	10	15	38	75
	P3.v2	6	89	53	91	157	287
	P3.v1	0	1	8	6	18	101

Table 10: Yearly distribution of the API specifications where the COLLECTION (P3) variants appear

GET/PUT/DEL/PATCH (P3.v1). The first variant in this collection is the one providing all of the GET, PATCH, PUT, and DELETE operations.

Although this variant includes most HTTP verbs and thus is the most expressive in terms of which operations clients can perform on collection items, it is far from being the most frequently used in practice. An example of use of this variant is in ID Vault API (Figure 51), where it appears 6 times.

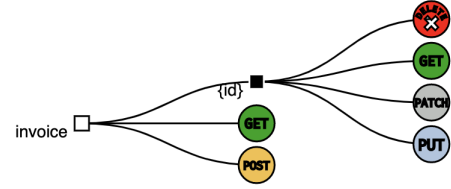Size: 8 — Occurrence: 328 — Distinct Labels: 159

Figure 29: Collection - GET/PUT/DEL/PATCH Variant (P3.v1)

GET/PUT/DEL (P3.v2). Fragments of this variant combine both the POST and GET operations on the collection. With more than one thousand occurrences, this variant (Figure 30) is the most occurring we have mined, not only within the variants of this collection but also among all the fragments having more than 3 distinct methods in their leaves. Several instances of this primitive can be found with different labels in the Apacta API (Figure 9).

Size: 7 — Occurrence: 1123 — Distinct Labels: 574
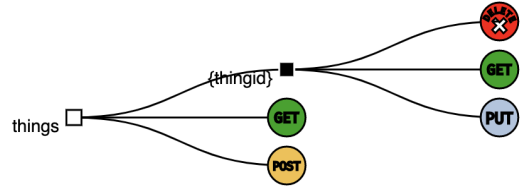
Figure 30: Collection - GET/PUT/DEL Variant (P3.v2)

GET/DEL/PATCH (P3.v3). This variant uses a PATCH operation instead of the PUT as in variant GET/PUT/DEL. Passman API (Figure 50) is an example of API where this variant appears.

Size: 7 — Occurrence: 233 — Distinct Labels: 139
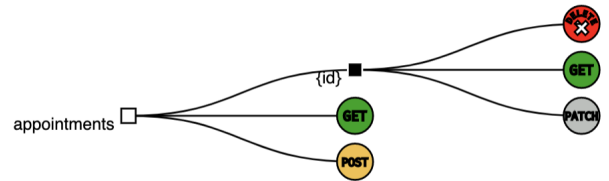
Figure 31: Collection - GET/DEL/PATCH Variant (P3.v3)

GET (P3.v4). This is the simplest variant of this collection. The client cannot perform any operation on the collection items, except to read their content. We have found examples of account collections, whose content cannot be modified by clients. Likewise, this is a common structure for long-running operations [15], which are started with a POST request used to transfer the input of the computation, while the status of the ongoing job and its result can be retrieved from the corresponding item.

Size: 5 — Occurrence: 323 — Distinct Labels: 168
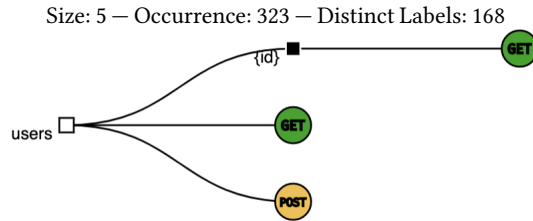
Figure 32: Collection - GET Variant (P3.v4)

PUT/DEL (P3.v5). This variant makes the client unable to individually read each item of the collection. However, it is possible to list them all, insert new items, delete or update them. Examples of such collections with unreadable items would contain simple items whose address indicating their identity and existence is sufficient to control their lifecycle (e.g., using the PUT operation to control the video or audio track playback). Likewise, to set the quantity of individual order line items or remove them from the order altogether one does not need to be able to retrieve any information about them. Also because such information can be fetched when enumerating the content of the entire collection.

Size: 6 — Occurrence: 169 — Distinct Labels: 84

Figure 33: Collection - PUT/DEL Variant (P3.v5)

GET/DEL (P3.v6). This variant only allows to read or delete individual collection items. This is one of the most frequently found variants, with a collection storing a wide variety of items. For example, once blog posts, comments, or questions are published, they cannot be updated but just removed. Likewise, it appears there is no need to update the ingredients of a recipe.
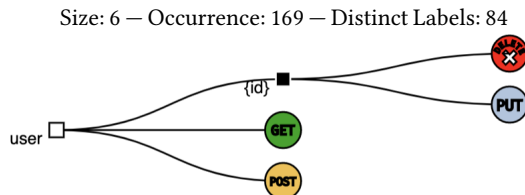
Size: 5 — Occurrence: 345 — Distinct Labels: 187
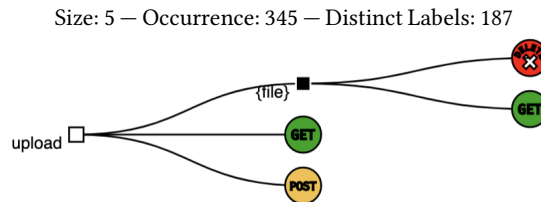
Figure 34: Collection - GET/DEL Variant (P3.v6)

DEL (P3.v7). This is a simpler variant, where it is possible to the client to list the elements of the collection and insert elements into it. Once the items have been added, it is only possible to remove them. In addition to bookings, this variant has been frequently used for collections of blog post comments, product reviews, or favorite bookmarks, whose content can be shown when retrieving the entire collection, but for moderation purposes, it may be necessary to be able to remove individual items.

Size: 5 — Occurrence: 201 — Distinct Labels: 87
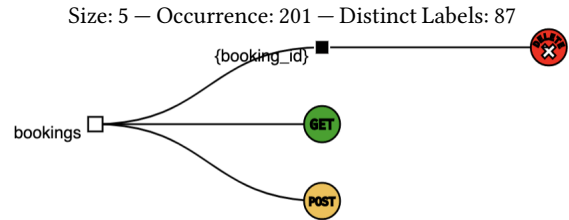
Figure 35: Collection - DEL Variant (P3.v7)

- **Collection Smells**

PUT-Only (P3.s1). ☐ Create without Delete

This smell provides only one operation to update individual items of the collection, but lacks the affordance for deleting individual items. This is used with collections of items whose state should be controlled by clients, for example to configure or simply switch on or off devices, gateways or services through a management API.

Size: 5 — Occurrence: 78 — Distinct Labels: 48
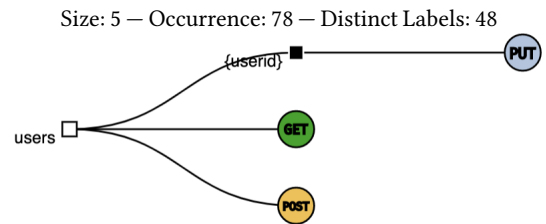
Figure 36: Collection - PUT-Only Design Smell (P3.s1)

GET/PUT (P3.s2). ☐ Create without Delete

Also, this smell does not allow clients to delete an item from the collection, however it allows them to insert items, read them and update them.

It has been used to design APIs which provide access to collections of users, user accounts, customers, employees, or withdrawals. These are resources which once they are created may need to be preserved forever for legal reasons, due to data preservation or retention regulations.

Size: 6 — Occurrence: 63 — Distinct Labels: 47
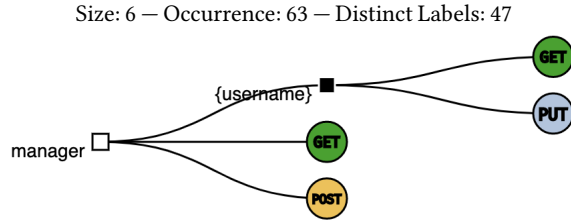
Figure 37: Collection - GET/PUT Design Smell (P3.s2)

Size: 9 — Occurrence: 74 — Distinct Labels: 52

Figure 39: Mutable Collection - GET/PUT/DEL/PATCH Variant (P4.v1)

4.4 Mutable Collection (P4)

Summary. Replace the content of the collection (PUT) or clear the entire content of the collection (DELETE)

Problem. How to let clients bring the whole content of the collection to a known state?

Solution. Add DELETE or PUT method to the container resource.
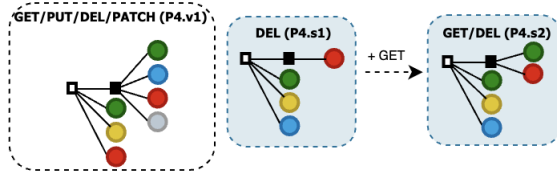
Figure 38: Mutable Collection (P4) Overview

	2015	2016	2017	2018	2019	2020
P4.s2	0	2	5	8	17	29
P4.s1	0	1	0	4	2	17
P4.v1	0	1	2	1	5	29

Table 11: Yearly distribution of the API specifications where the Mutable Collection (P4) variants and smells appear

- **Mutable Collection Variants**

The two variants with PUT do not really seem to be used to replace the content of the entire collection (only a few exceptions). Instead, the PUT method is used to edit or update individual items, addressed by a query parameter that identifies the item to be replaced. (Query parameters are not shown in the figure).

***GET/PUT/DEL/PATCH (P4.v1)*.** This variant provides only a delete operation on the collection. This variant occurs 74 times. It provides besides the DELETE and GET operations on the collection items, also PUT and PATCH operations. The second label for all the labels sequences is $\{name\}$ except one sequence $S = \{boards , \{id\}\}$

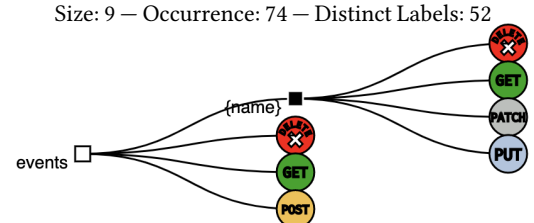For verifying the purpose of use of the DELETE method over the collection, we have extracted the descriptions associated with that method in the OpenAPI specification of the APIs where that fragment occurs (Table 17). Indeed, in this case, the DELETE operation is used to delete to the whole collection. This "DELETE all" variant could be promoted to a separate primitive named "Erasable Collection".

	P2.v1	P2.v2	P2.v3	P2.s2	P2.s1
account	1	0	6	0	0
annotations	3	0	0	0	0
annotationsets	3	0	0	0	0
assets	0	0	0	3	0
bookings	0	1	0	1	0
buy	0	0	2	0	0
cart	0	1	2	0	0
category	1	0	0	0	1
collections	0	0	3	0	0
comment	0	1	0	1	5
courses	1	4	0	0	0
datapointers	3	0	0	0	0
deployments	0	1	11	0	0
disease	0	0	5	0	0
employee	0	1	1	0	1
entries	1	0	2	0	0
experiments	0	0	2	0	0
files	0	1	6	0	0
form_fields	0	0	6	0	0
governance	0	0	2	0	0
hub	0	2	0	1	0
images	0	0	11	1	0
individuals	3	0	0	0	0
inventory	0	0	2	0	0
item	1	1	3	0	0
jobs	0	0	1	2	0
labels	0	0	1	5	0
media	0	4	1	0	0
messages	2	1	9	0	0
objectstores	0	0	0	0	8
order	4	78	1	1	1
policy_keys	0	0	3	0	0
post	0	0	2	1	1
productos	0	0	0	0	4
products	2	0	2	0	0
provider	2	0	0	0	1
read	0	0	5	0	0
register	11	0	0	0	0
student	1	1	1	0	0
subscriptions	1	0	0	2	0
target	0	0	5	0	0
task	1	0	1	0	1
todo	1	12	1	0	0
token	0	0	1	11	0
traces	0	0	4	0	0
user	114	5	11	3	7
wall_comments	0	0	6	0	0

Table 12: Mutable Collection (P4) – Labels found in each variant/smell

- **Mutable Collection Design Smells**

DEL (P4.s1). ▭ Ambiguous PUT This structure provides both read and append operations over the collection, in addition to a PUT operation. While the only operation that the client can perform over the collection items is a delete. For verifying the real purpose behind having a PUT operation over the collection resource we extracted in Table 20 the descriptions associated with this method in the OpenAPI documents. The text shows that, in reality, in almost all cases, the PUT is used to update an item of the collection. The address of the item is provided as a request parameter, as opposed to using the resource path as with most other primitives. Only in a few cases, it is actually used as one would expect, for updating the whole collection with a single batch operation to replace its content (Example D-13).
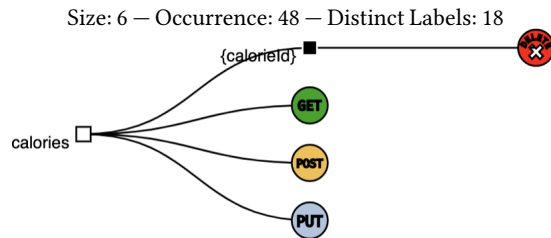
Size: 6 — Occurrence: 48 — Distinct Labels: 18

Figure 40: Mutable Collection - DEL Design Smell (P4.s1)

GET/DEL (P4.s2). ▭ Ambiguous PUT

The fragments of this variant combine all of GET, PUT and POST operations on the collection. An example of the most occurring fragment in this variant is in Figure 41. This fragment occurs 102 times with 56 distinct Label Sequences.

Also, in this variant the PUT method on the container resource is used mostly to update the content of individual items (Table 18), thus leading to some ambiguity as it should have been associated with the collection item resource.

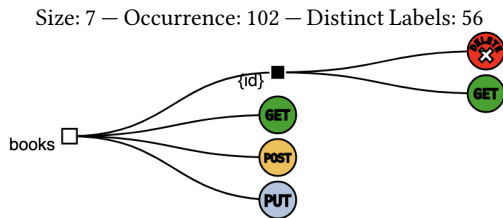Figure 48 show an example of API where this variant is used twice.

Size: 7 — Occurrence: 102 — Distinct Labels: 56

Figure 41: Mutable Collection - GET/DEL Design Smell (P4.s2)

5 FROM PRIMITIVES TO LARGER STRUCTURES AND API RESPONSIBILITY PATTERNS

This section gives two examples of how the mined primitives presented in the previous sections can be used during API design and API reviews. First, we discuss primitive composition. Next, we briefly outline how the structural primitives from this paper relate to previous work on API design patterns and interface description languages.

5.1 Composing Primitives

The basic collection primitives can be composed to form larger API structures in two ways:

(1) Unrelated collections can be added to the API by adding the corresponding container resource on the same level as shown in the fragments of Table 13;
(2) Related collections can be nested inside one another, by adding a sub-container resource within each item of the main collection, as shown in the fragment of Table 14.

In general, we found that both side-by-side composition and nesting can be used together in the same API. The Invostra API shown in Figure 45 is an example of an API entirely composed of two primitives.

5.2 Relation to Architectural Patterns and Interface Description Languages (IDLs)

In the patterns community, technology- and platform-neutral interface representation and service design patterns have been mined and published. The Microservice API Patterns (MAP) language [29], for instance, focuses on the design of remote APIs — including but not limited to service-oriented architectures. MAP has two categories that complement the API primitives and fragments carved out in this paper, *structure* [30] (of request and response message representations, not HTTP resource tree structures as covered in this paper) and architectural *responsibility* [26].

The HTTP methods found in the resource trees in Sections 2, 3, and 4 map to the MAP language as this:

- HTTP GET methods are "Retrieval Operations"[26].
- HTTP POSTs can be "State Creation Operations" but also "State Transition Operations" (partial update variant) [26].
- HTTP PUTs are "State Transition Operations" (full replacement variant).
- PATCHes correspond to "State Transition Operations" (partial update variant).
- DELETE methods are represented as variants of State Transition Operations.

The remaining operation pattern from MAP, "Computation Function", can be mapped to HTTP GET (if its request parameters are simple) or POST (if request parameters are complex).

The collection primitives that we derived from URI structures in Section 4 correspond to the MAP endpoint pattern "Information Holder" and its specializations[27]:

- "Master Data Holders" expose many GET retrievals and only a few bulky create POSTs and update PUTs. They usually are enumerable, and also appendable (at least for certain clients).
- "Operational Data Holders" typically are Enumerable and Appendable Collections, often also Mutable Collections.
- "Reference Data Holders" are read-only and therefore Enumerable Collections.
- "Data Transfer Resources" are Mutable Collections decoupling multiple application clients.
- A set of related "Link Lookup Resources" forms a Collection as well; each item in such collection is mutable and so is the entire collection.

Table 13: API Fragments Composing the Read-only Collection and the Collection primitives side by side
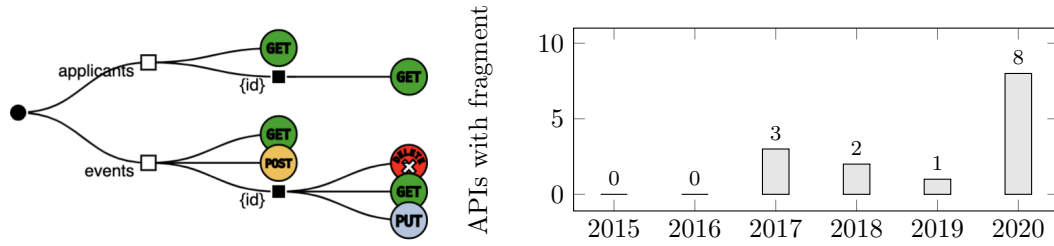
Size: 12 — Occurrence: 33 — Distinct Labels: 12

Label clusters of sample known uses:
(O: Occurrence, C:Cohesion)

Cluster	$Label_1$	$Label_2$	$Label_3$	$Label_4$	O	C
C1	applicants	{id}	events	{id}	2	8.33
	users	{id}	teams	{id}	1	8.16
	users	{id}	tasks	{id}	1	8.33
	gst	{gst_id}	base_slice_des	{id}	1	2.47
	application−templates	{id}	applications	{id}	1	4.08
C2	contact_types	{contact_type_id}	contacts	{contact_id}	6	8.04
	form_templates	{form_template_id}	forms	{form_id}	6	8.16
	time_entry_intervals	{time_entry_interval_id}	time_entry_types	{time_entry_type_id}	6	8.04
	time_entry_value_types	{time_entry_value_type_id}	users	{user_id}	6	8.04
C3	resources	{location}	function	{func_id}	1	2.66
C4	cookings	{uuid}	programs	{uuid}	1	8.04

Table 14: API Fragments Composing the Read-only Collection primitives with nesting
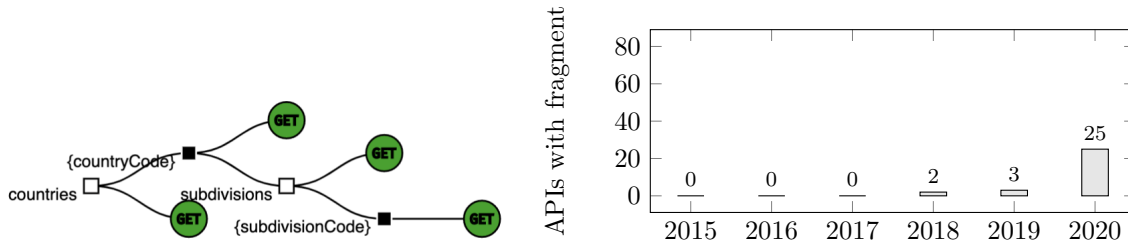
Size: 8 — Occurrence: 32 — Distinct Labels: 13

Label clusters of sample known uses:
(O: Occurrence, C:Cohesion)

Cluster	$Label_1$	$Label_2$	$Label_3$	$Label_4$	O	C
C1	companies	{company_id}	integration_feature_settings	{integration_feature_setting_id}	5	6.66
	identity−classes	{identityClassID}	levels	{identityLevelID}	1	7.44
C2	BRK_waardelijsten	{waardelijstidentificatie}	waarden	{code}	2	4.98
	tabellen	{tabelidentificatie}	waarden	{code}	2	5.57
C3	components	{component}	resource−managers	{resource−manager}	2	6.05
C4	stations	{stationId}	history	{columnName}	1	5.65
C5	entities	{entityName}	views	{viewName}	3	7.64
C6	countries	{countryCode}	subdivisions	{subdivisionCode}	1	8.05

Microservice Domain-Specific Language (MDSL)[10] is an emerging abstract service contract language that exposes the Microservice API (MAP) patterns used above as *decorators*. As an example, let us model the example from Section 2 in MDSL (including the data contracts specifying request parameters and response representations, as well as error reports):

```
API description ApactaAPI version "v1"
overview "See previous paper sections and APIs.guru"

data type CityCollection {
  "data":City*,
  "pagination":PaginationDetails,
  "success":Metadata<bool>}

data type City {
  "created":Metadata<string>,
  "deleted":Metadata<string>,
  "id":ID<string>,
  "modified":Metadata<string>,
  "name":Data<string>,
  "zipCode":Data<int>}

data type PaginationDetails {
  "count":Metadata<int>,
  "current_page":ID<string>,
  "has_next_page":Metadata<bool>,
  "has_prev_page":Metadata<bool>,
  "limit": Metadata<int>,
  "page_count":Metadata<int>}

data type ErrorNotFound {
  "data": {
    "code":Data<int>,
    "message":Data<string>,
    "url":Link},
  "success":Metadata<bool>}

endpoint type CityEndpoint
 serves as REFERENCE_DATA_HOLDER and COLLECTION_RESOURCE
exposes
  operation getListOfCities
   with responsibility RETRIEVAL_OPERATION
    expecting payload "zipCode":Data<string>?
    delivering payload CityCollection
    reporting error searchFailed "404": ErrorNotFound
  operation getCityDetails
   with responsibility RETRIEVAL_OPERATION
    expecting payload "cityId":ID<string>
    delivering payload City
    reporting error noSuchCity "404": ErrorNotFound
```

In MDSL, such abstract service contract, possibly discovered via the the stepwise fragments and primitives mining approach established in previous sections, can be translated to not only to HTTP but also gRPC, GraphQL schemas, and other IDLs. The translation to an HTTP resource API is not straightforward and therefore benefits from an explicit binding (defaults exist, but usually are not

sufficient a) to establish resource trees that contain parametric path segments and b) to map endpoints that contain more operations than the unified verb interface of HTTP can bear):

```
API provider CityEndpointProvider
offers CityEndpoint
at endpoint location "http://tbc.tbc.tbc:8080"
via protocol HTTP binding
resource CityCollection at "/cities"
  operation getListOfCities to GET
    element "zipCode" realized as QUERY parameter
  report searchFailed realized as 404 with "Not found"
resource CityDetails at "/cities/{cityId}"
  operation getCityDetails to GET
    element "cityId" realized as PATH parameter
  report noSuchCity realized as 404 with "Not found"
```

The above specification snippets merely demonstrate that it is feasible to model HTTP resource APIs in MDSL and to leverage the primitive names from previous sections as well as MAP decorators while doing so. A thorough introduction to MDSL can be found in [10], and the language reference is available at https://microservice-api-patterns.github.io/MDSL-Specification.

6 RELATED WORK

6.1 Model Clustering

In our work, we analyze and mine a large data set of real-world OpenAPI specifications. We extracted selected building blocks from these specifications, focusing on the resource URI tree structures and HTTP method verbs in particular. We cluster these model blocks both structurally and semantically. Hence we have the same goals as the authors of [4], but in a different context. This paper presented an approach for clustering models using n-grams in order to incorporate the structural context of the models in this task, and also to see the impact of using n-grams on the resulted clusters. They applied the approach on a dataset of Ecore metamodels collected from AtlanMod Metamodel Zoo using different sizes of n-grams, where they found that the clustering accuracy does not increase monotonically along with increasing the size of the n-grams. In our case, we involve all the label sequences extracted from fragments models in the semantic clustering task, except the names of the method, as we consider them part of the structure rather than being part of the semantic context.

6.2 Structural Analysis of Web APIs

Many works tried to statically extract structural characteristics and functional properties of APIs from their textual descriptions (both informal or formal documentation). Similar to our approach, in [8], the authors performed a structural analysis over 286 real-world publicly available API specifications (retrieved from apis.guru). The authors defined a set of quantitative metrics related to the resources and the HTTP methods supported by the API and identified the challenge of extracting recurring API fragments.

An earlier study [12] manually collected and analysed a set of 222 non machine-readable publicly-available APIs descriptions gathered from programmableweb.com by selecting randomly APIs from each category, hence covering 18% of the APIs listed on the website at that time.

From it, Maleshkova. M et al. [12] extracted some metrics such as type of Web API, input parameters, output formats, invocation details and if the API has complementary documentation. While [8], and [12] focused more on metrics extraction and providing static results about the sample under study, the authors of [13] performed an in-depth analysis in order to detect five REST design patterns and eight anti-patterns. To do so, they defined and implemented detection heuristics. Likewise, the authors of [19] also focused on verifying the compliance of REST APIs with REST constraints. However, the analysis presented in [19] was performed over a large data set of 78GB of HTTP requests corresponding to one full day of Mobile Internet traffic, collected by Italy's biggest Mobile Internet provider. This work reached the conclusion that only a few of the analyzed APIs comply with the best practices and constraints of the REST architectural style.

[16] is another work questioning the design quality of web APIs, focusing mainly on REST APIs for cloud computing. They defined a catalog of 73 best practices in the design of REST APIs, in terms of understandability and reusability, starting from a literature review. And they applied it over a set of well-known APIs in the Cloud Computing area, where they found that Google Cloud follows 66% (48/73), OpenStack follows 62% (45/73), and OCCI 1.2 follows 56% (41/73) of their best practices.

While these works focus on evaluating the quality of the design of different Web APIs collections, our approach is one of the first systematic and quantitative studies to recover common structural design decisions adopted by Web APIs creators.

More patterns for various styles of distribution have been mined previously, including Messaging Patterns [9], Remoting Patterns [23], Patters of Enterprise Application Architecture [6], and Service Design Patterns [5]. Many of these works discuss APIs, interface representation, and service design in the particular field of distributed system technologies and architectures they focus on.

7 THREATS TO VALIDITY

Basing empirical studies on resources collected for public repositories shared publicly on version control system is prone to error, because of the fact that not all what is shared is accurate. During our analysis, we encountered 2534 (38,28%) invalid OAS documents, which contain errors related to the conformity to the OpenAPI metamodel. In this work, we transformed the exact content of the OAS models into our API tree model, which also reflects the errors that may be encountered in the OAS document. Such errors may have reduced the number of occurrences of some structures, as the ones containing mistakes can't be matched to other error-free structures.

The study performed in this work is based on a data set of OAS documents mined from GitHub. To avoid having a biased data set full of duplicates, we only inserted in the OAS DB distinct documents. However, there can be forks that did not introduce changes to the structure of the API, which could have inflated the values of the occurrence and the popularity metrics of some fragments.

While not all OAS documents we found on GitHub describe Web APIs offered in production, including partially developed artifacts in our analysis can still provide evidence of reuse of common API fragments.

8 CONCLUSION

In this paper, we presented a data-centric pattern mining approach. We applied it to find recurring primitive structures within Web API descriptions.

To do so, we extracted recurring fragments from a large collection of OpenAPI specifications gathered from open source repositories. Reflecting the hierarchical nature of HTTP-based APIs, these fragments are represented as trees. These trees are built out of resource identifiers; they can be traversed to obtain all paths that are present in the original OpenAPI specification. The leaves of the API trees refer to the HTTP methods that invoke the corresponding operations.

From a population of thousands of fragments, we selected those that a) frequently occur, b) have a relatively small size, and c) are centered around the notion of resource collection. As shown in Figure 14, we distinguish the following cases a) collections only offer operations on their items or on the collection level as well, b) their content can be enumerated, new items can be created, and/or both enumeration and creation are supported, and c) batch removal and update are provided.

For every primitive, we presented a selection of variants together with the corresponding label clusters and, in some cases, descriptions associated with the operations. A few variants can be also seen as design smells, for instance, if they use the HTTP method semantics incorrectly or inconsistently.

Our results are a collection of pattern primitives, which can be (and have been) composed to build larger API structures. For instance, this becomes evident when connecting the syntactical patterns that we mined here automatically with semantic architectural patterns previously mined manually by knowledge engineers[26, 27]. We merely gave the first examples of primitive composition in this paper, and only outlined these connections; in our future work, we plan to investigate these topics more thoroughly.

ACKNOWLEDGMENTS

APPENDICES

We attach to this paper four appendices with additional information regarding:

- Tree visualizations of some real-world known uses of the pattern primitives variants in Appendix C, using the notation described in Table 1.
- API Fragments overview in Appendix B, we plot the different correlations: Fragment size vs. Occurrences, Number of unique label combinations vs. Occurrences and Fragment size vs. Number of unique label combinations.
- Textual descriptions of some HTTP methods in selected variants in Appendix A, extracted from the original specification of the APIs where a specific variant or smell appears.
- Most frequent labels, for each primitive variants and smells in Appendix D.

A EXTRACTED DESCRIPTIONS OF HTTP METHODS

This appendix contains tables for some selected methods description, used to verify the Design Smells discovered in our study. We detect a design smell when the natural language descriptions associated with the API feature are not consistent with the standard semantics of the chosen HTTP method.

A.1 Enumerable Collection

Table 15: Extracted description of the POST method for Enumerable Collection - GET/POST Design Smell (P1.s1)

(1) Generates customized software development kit (SDK) and or tool packages used to integrate mobile web or mobile app clients with backend AWS resources
(2) "Write a range of table elements"
(3) Alert about something
(4) Builds templated versions of the challenge- Uses the flag format and seed to template out a new version of the challenge This may take a signficant amount of time-
(5) Create a deployment request
(6) Create a new user in system
(7) Generate token for valid user
(8) Perform pruning on input resource name
(9) Post message by username- Creates a message with the username as author
(10) Save a new revision of a page given in HTML format
(11) Set automation state- Set automation state for the given automation type
(12) Sets the value of a float variable
(13) Sets the value of a string variable
(14) Sets the value of an integer32 variable
(15) This endpoint returns the result of executing this operation
(16) This endpoint returns the result of executing this test
(17) Upload an Attachment- Upload an Attachment-

Table 16: Content of the description field of the DELETE method of the variant GET/PUT/DEL (P1.v4)

(1) Close an existing position
(2) Delete Link- Will not delete the target object
(3) Delete a contact device for a user Delete a contact device for a user
(4) Deletes the given device, and invalidates any access token assoicated with it
(5) Delete a directory tenant under a resource group
(6) Delete a node- Remove the node identified by id A node can only be deleted if it is currently offline
(7) Delete a node- Remove the node identified by id A node can only be deleted if it is currently offline and does not host any master deployments
(8) Delete file- Delete file uploaded to a project from wall post or form
(9) Delete maintenance configuration
(10) Delete mock definition
(11) Delete snapshot repository- Deletes a snapshot repository configuration by name
(12) Delete the scheduled override assignment- Delete the scheduled override assignment
(13) Deletes a policy definition at management group level
(14) Deletes a policy definition
(15) Deletes a product package
(16) Deletes a server communication link
(17) Deletes a user from the list of registered users
(18) Deletes an acquired plan
(19) Deletes an existing server Active Directory Administrator
(20) Deletes single user
(21) Deletes specified file container- Delete an existing file container-
(22) Deletes specified quota- Delete an existing quota
(23) Deletes the MariaDB Server key with the given name
(24) Deletes the MySQL Server key with the given name
(25) Deletes the PostgreSQL Server key with the given name
(26) Deletes the log profile-
(27) Deletes the specified Azure key vault-
(28) Deletes the specified application security group
(29) Deletes the specified public IP address-
(30) Remove a CIDR Map
(31) Remove a Geographic Map
(32) Remove a Property
(33) Remove a Resource
(34) Remove a single task
(35) Remove an episode vote
(36) The operation to delete a container service
(37) Unfollow a network
(38) Unfollow a person
(39) Unfollow a show
(40) Unfollow a webchannel
(41) Unmark an episode
(42) delete an Ad- you must own the Ad and be logged in to delete an Ad. Deleting an Ad will also erase al pictures uploaded to the API linked to it

A.2 Mutable Collection (P4)

Table 17: Example of extracted description for the DELETE method of the variant GET/PUT/DEL/PATCH (P4.v1)

(1) Delete a collection of CDI objects
(2) Delete a collection of CDIConfig objects
(3) Delete a collection of DataVolume objects
(4) Delete a collection of VirtualMachine objects
(5) Delete a collection of VirtualMachineInstance objects
(6) Delete a collection of VirtualMachineInstanceMigration objects
(7) Delete a collection of VirtualMachineInstancePreset objects
(8) Delete a collection of VirtualMachineInstanceReplicaSet objects
(9) Delete all
(10) delete collection of AerospikeCluster
(11) delete collection of AerospikeNamespaceBackup
(12) delete collection of AerospikeNamespaceRestore
(13) delete collection of Alb
(14) delete collection of AppBinding
(15) delete collection of AuditRegistration
(16) delete collection of Branch
(17) delete collection of Bundle
(18) delete collection of Certificate
(19) delete collection of Cluster
(20) delete collection of ClusterAuthInfoTemplate
(21) delete collection of ClusterInfo
(22) delete collection of ClusterUserAuth
(23) delete collection of ConfigMap
(24) delete collection of Credential
(25) delete collection of Dashboard
(26) delete collection of Event
(27) delete collection of Ingress
(28) delete collection of KubeDBOperator
(29) delete collection of Message
(30) delete collection of MessagingService
(31) delete collection of Pipeline
(32) delete collection of PullRequest
(33) delete collection of Repository
(34) delete collection of S2iBuilder
(35) delete collection of S2iBuilderTemplate
(36) delete collection of S2iRun
(37) delete collection of Secret
(38) delete collection of Snapshot
(39) delete collection of StashElasticsearch
(40) delete collection of StashMariaDB
(41) delete collection of StashMongoDB
(42) delete collection of StashMySQL
(43) delete collection of StashPerconaXtraDB
(44) delete collection of StashPostgres
(45) delete collection of StashRedis
(46) delete collection of TFJob
(47) delete collection of Tag
(48) delete collection of Workflow

Table 18: Descriptions for the PUT method of variant GET/DEL (P4.s2)

(1) Actualiza un evento
(2) Actualiza un libro
(3) Actualizar inmueble - Actualizar un inmueble de la API
(4) Edit a floor
(5) Edit a stack
(6) Return the updated user
(7) Update Book details - Update Book details
(8) Update CampaignRecipient
(9) Update CampaignSettings
(10) Update ListCampaignDefaults
(11) Update ListContact
(12) Update a existing FX order not matched yet, on the market place
(13) Update an existing FX order
(14) Update a list - Update a List
(15) Update a song - Update a song
(16) Update an event - Update an event
(17) Update an existing account
(18) Update an existing blog - API Endpoint to update a blog
(19) Update an existing building MAP in database
(20) Update an existing conversation
(21) Update an existing deluge
(22) Update an existing payment-instruction not settled yet
(23) Update an existing payment-instruction
(24) Update an existing rule
(25) Update an existing scenario
(26) Update an existing skill
(27) Update client information
(28) Update collaborator
(29) Update role
(30) Update the microapp - Update the microapps
(31) Update theme - Update theme
(32) Update user
(33) Updates a given field for an attack with a certain ID
(34) Updates and existing agent class to agent manifest mapping configuration
(35) replaceGroups - Replaces user's roles with the submitted ones
(36) sending a draft mail to a user
(37) update Category - Update Category
(38) updateCustomer
(39) updateProduct
(40) updates a track from the system - Updates a track from the system

A.3 Appendable Collection (P2)

Table 19: Description and summary of the GET method in GET/PUT/DEL (P2.v1)

(1) Find Tracks by ID- Returns a single Tracks
(2) Find ad by ID- Returns a single Ad
(3) Find ad_html_meta by ID- Returns a single AdHtmlMeta-
(4) Find client by ID- Returns a single client
(5) Find course by ID- Returns a single course
(6) Find item by ID- Returns a single Item
(7) Find order by ID- Returns a single order
(8) Find pet by ID- Returns a single pet
(9) Find product by ID- Search one product by id
(10) Find provider by ID- Returns a single provider
(11) Find user- Returns a user
(12) Finds News by Id- Returns a single news
(13) Get Address by ID
(14) Get Student By Name- Get Student Details by name
(15) Get Usage by id
(16) Get a Client Registration for a given Client ID
(17) Get a Distribution- Get a Distribution
(18) Get a client by way of Client ID
(19) Get a project by project_id
(20) Get a single message
(21) Get a specific city
(22) Get a user by ID
(23) Get a user- Return a json object of the user
(24) Get an Assessment object
(25) Get bucketlist with given ID for loggedIn User
(26) Get details of an Order
(27) Get infos about a specific exam- Returns the exam id
(28) Get match-
(29) Get one Product with specified ID
(30) Get provider by user code
(31) Get region by id
(32) Get scotch by id
(33) Get table
(34) Get team
(35) Get user by id- Get the user information by its id
(36) Get user by user id
(37) Get user by user name
(38) Gets Business Partner Object
(39) Gets an annotation Caller must have READ permission for the associated annotation set
(40) Gets an annotation set Caller must have READ permission for the associated dataset
(41) Gets the details for an order
(42) Look up a user by their user id
(43) Obter um momento
(44) Retrieve the information associated with a signin record Retrieve the information associated with a signin record
(45) Return a Question by ID- Returns a single Question object
(46) Returns a nomination based on a single ID- Returns the nomination identified by 'nominationId'
(47) returns a single entry

A.4 Collection (P3)

Table 20: Extracted description of the PUT method of variant DEL (P3.v7)

(1) Adds the secrets specified in the payload The payload must be a JSON object where the keys are the secret names and the values are the secret values If a secret already exists, it is overwritten
(2) Edit a library
(3) Edit product This method allows you to edit existing product
(4) Modifica un usuario- Modifica un usuario por su identificador
(5) Overwrites the secrets for the specified system The payload must be a JSON object where the keys are the secret names and the values are the secret values
(6) Persist plugin metadata information
(7) Update a given beverage Requires ADMIN role- The beverage must be already existent Return value is the updated and stored data
(8) Update a snapshot schedule
(9) Update an existing role
(10) Update existing cloud backup schedule
(11) Update item in calories list for user
(12) Update user with give ID
(13) Updates an existing attribute- Updates an existing attribute
(14) Updates specified storage policy
(15) setProjectAgentPools
(16) update a tenant- Update a tenant

B API FRAGMENTS OVERVIEW

To give an overview of the API Fragments, we measure their size, their number of occurrences, and the number of unique label combinations they have.

Figure 42 shows that the larger the fragment, the less likely it is to reoccur multiple times. The largest fragments occur only once. Our constraints in the search of API fragments that can be used as pattern candidates is to both avoid unpopular fragments while also ensuring to find "interesting", large-enough fragments.

While it is difficult to see in the chart, each dot represents not only a fragment of the given size and occurrences, but since there can be multiple fragments in the same coordinates we also color each dot by the number of fragments found in that position. Most of the fragments crowd into the origin area and have few repeated occurrences. While the chart shows there is a lot of variability in the structure of APIs, it also shows that – this is an extreme case – there are four small fragments that reoccur more than 3000 times.

More in detail, we are interested to observe the actual labels on the API fragment since they give an indication of the API semantics by giving names to its features. Labeled API fragments provide potential known uses for the candidate patterns.
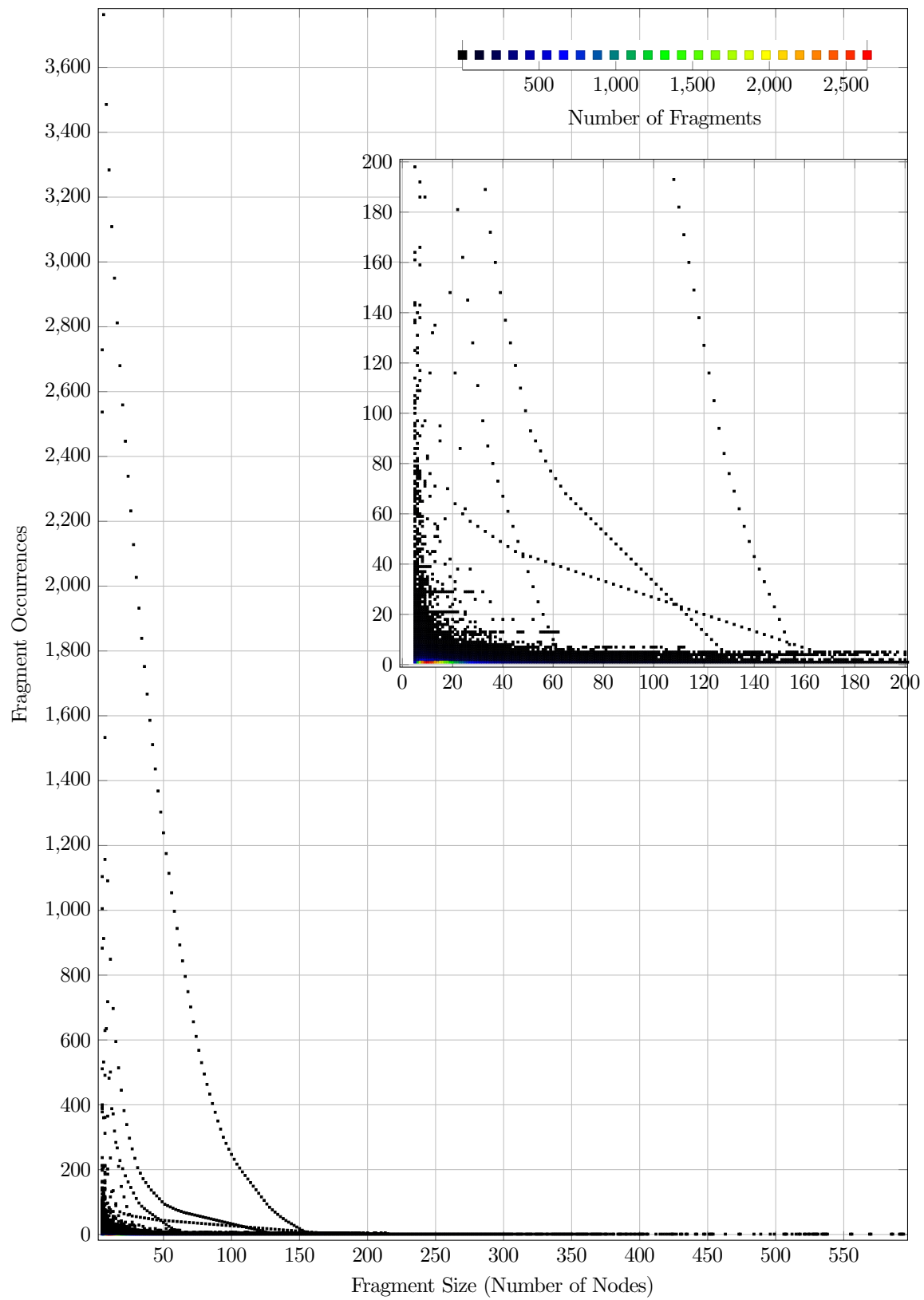
Figure 42: API Fragments Overview (Fragment Size vs. Occurrences)
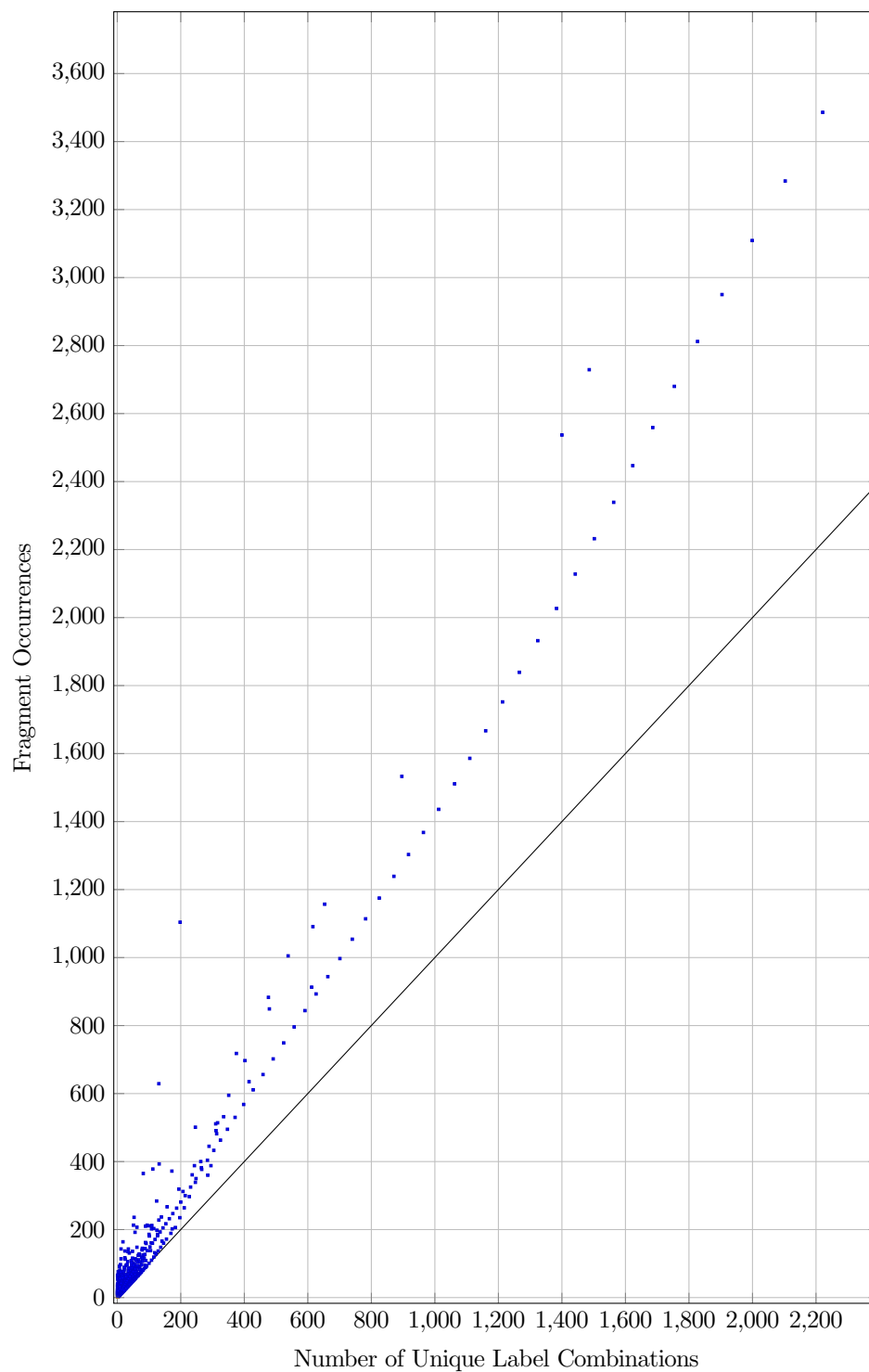
Figure 43: This plot shows the relationship between how many times API fragments occur and how many unique label combinations they have. Some fragments occur across thousands of APIs with thousands of unique label combinations. The vertical distance of a dot representing a fragment from the Y=X line indicates how many times the fragment reoccurs with exactly the same label combination. This may be due to redundancy present in the API collection (some API descriptions have been cloned or forked) or also because a fragment happens to be frequently used with the same labels.
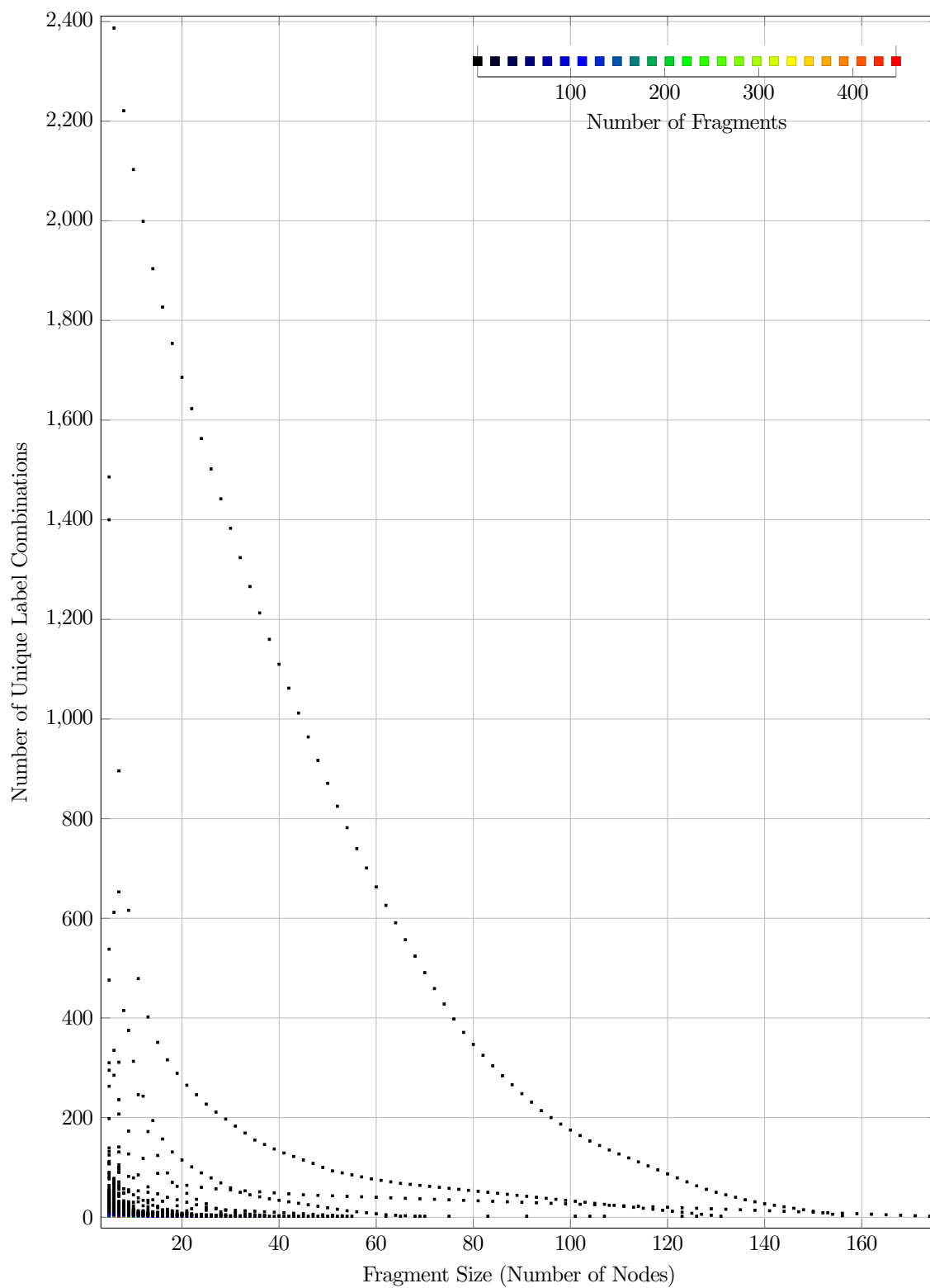
Figure 44: This plot compares the size of the fragment with the number of unique label combinations it appears with. Compared to the raw number of occurrences, the Y-axis shrinks from 3600 down to 2400. The figure does not include fragments with a single occurrence, thus the X-axis range is also reduced accordingly.

C API TREE VISUALIZATIONS

This appendix contains visualization of some real world APIs selected as examples for this study. They are the APIs where some variants and smells of the primitives we have described in Section 4 can be found with high occurrences.

C.1 Invotra API

The Invotra API[2] can be simply see and a combination of GET/PUT (P3.s2) (occurs twice) and PUT/DEL (P1.v3). It is an example in which we can see both ways of primitives composition described in Section 5.1
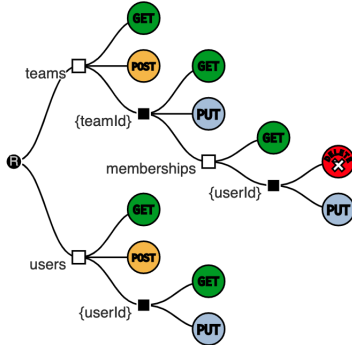
Figure 45: Invotra API Tree Visualization

C.2 TvMaze user API

In the TvMaze API[3] we can find six occurrences of Enumerable Collection - GET/PUT/DEL Variant (P1.v4), combined with one read operation in the path /vote/shows.
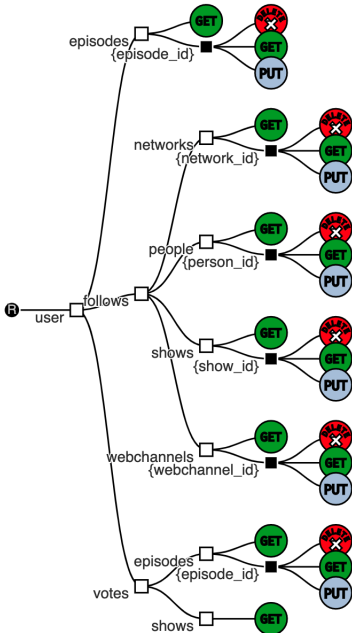
Figure 46: Tree visualisation for the TVmaze user API

[2]https://github.com/invotra/api
[3]https://static.tvmaze.com/apidoc/

C.3 Columba API

The Columba API [4] uses tree instances of the GET (P1.v1) variant and one of the GET/PUT (P1.v2) variant.
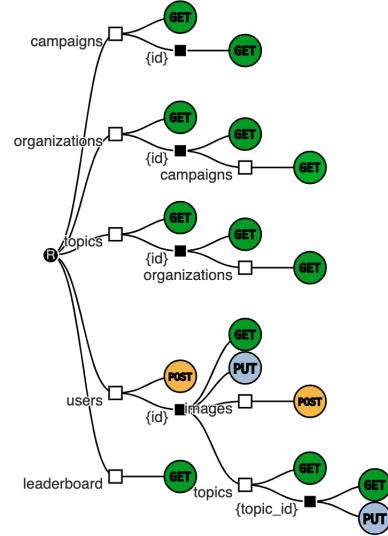
Figure 47: Tree Visualization of Columba API

C.4 AnyPay API

AnyPay service targets parents with children doing payments. It is an example of usage of the GET/DEL (P4.s2) variant.
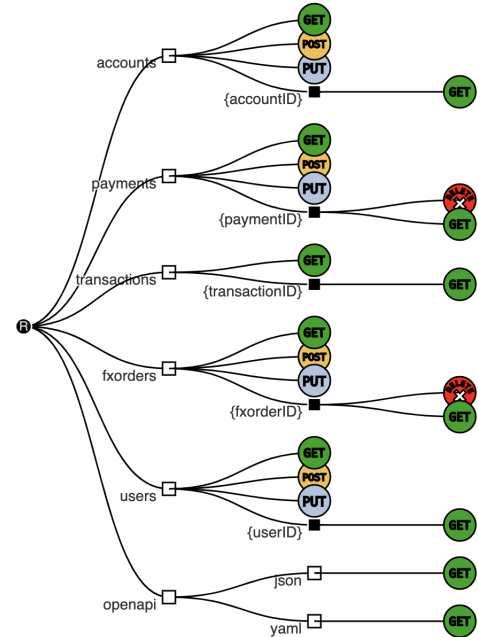
Figure 48: Tree visualization of AnyPay API

[4]https://github.com/columbasms/columbasms.github.io

C.5 API for the COVID-19 Tracking QR Code Signin Server

This is an API for the COVID-19 Contact Tracing QRCode Signin Server. It combines the GET/DEL (P2.v2) and GET/PUT/DEL (P2.v1) variants of the APPENDABLE COLLECTION (P2) primitive with a set of paths with a unique method (POST or GET).
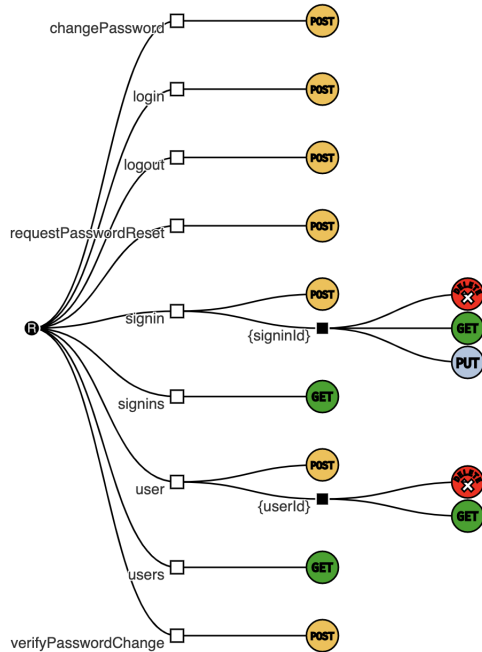
Figure 49: Tree visualization of an API for the COVID-19 Tracking QR Code Signin Server

C.6 Passman API

The Passman API [5] combines all of the GET/DEL/PATCH (P3.v3) and GET/DEL (P2.v2).
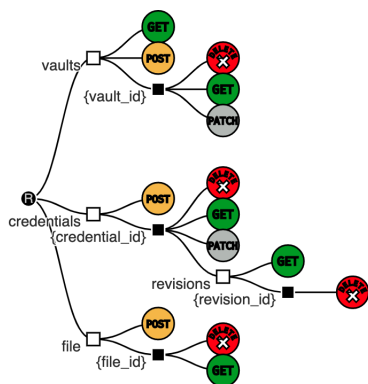
Figure 50: Tree Visualization of Passman Developers API

---
[5]https://github.com/nextcloud/passman#api

C.7 ID Vault API

This is an API example where the Collection - GET/PUT/DEL/-PATCH Variant (P3.v1) in appearing several times, combined with one use of GET/PUT/DEL (P3.v2) variant.
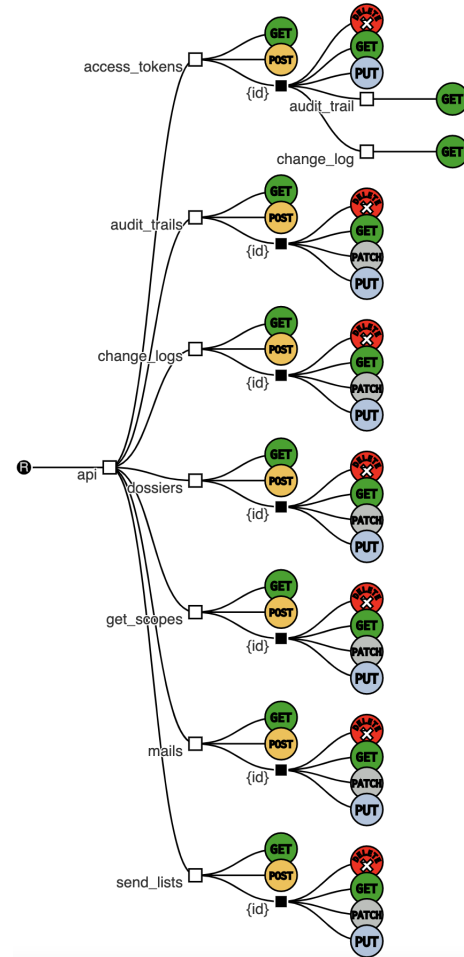
Figure 51: Tree Visualization of ID Vault API

D LABELS USAGE

The goal this Appendix is to provide an additional overview about the labels usage through a set of guidance heat maps to help in plotting the most used labels for each structural primitive.

We calculated the occurrences of each label in each structural primitive. The labels are sorted first by their total number of occurrences, then alphabetically so that it is easy to find at the top of the table which are the most used labels in each primitive. The most used label for a specific variant is found by looking at the row which has the darkest color in the variant's column.

To know in which variant a specific labels sequences is used the most, it is enough to horizontally scan the row for the label looking for the highest value, ignoring the last column in which the total number of occurrences across all variants is reported. The heat maps only include the representative labels that we obtain using our labels merging approach described in Section 3.1.2.

We merged labels sequences with common container labels such as the ones in Figure 52. These labels are used in different variants of
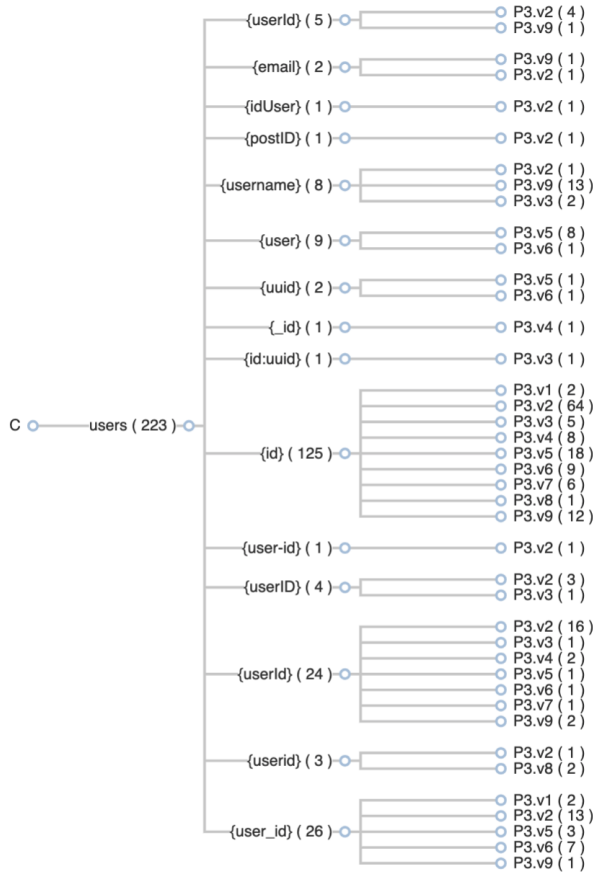
Figure 52: Label sequences with container label "users" in Collection primitive

the COLLECTION (P3) primitive and they share the same meaning, thus we decided merging them in order to have concise guidance tables. In our labels processing approach, we ignore the case of the container resource label (e.g : *dataPointer* is equivalent to *datapointer*). We also remove all the special characters and the spaces (e.g: *dataPointers*, *data−pointers*, *data_pointer* are considered equivalent). Moreover, ignore the singularity and plurality of the labels (e.g: *data_pointer* is equivalent to *dataPointers*). In the example of Figure 52 the label *users* originally appears in different formats (e.g: *user*, *Users*, *User*), our merging algorithms pick the most occurring form as a representative label of all the forms. The reason behind this clustering is to give more insight about the labels usages and merge the ones that represent similar concepts in order to avoid redundancy.

D.1 Enumerable Collection (P1)

In this primitive, the GET (P1.v1) is the one that appears with the highest number of distinct labels. We notice that most of the labels are only used by this variant, while other few ones are used also by the other variants (occurs 202 times).

|	P1.v1	P1.v2	P1.v3	P1.v4	P1.s1	P1.s2	OCC
operations	51	0	0	0	5	0	56
users	29	1	4	11	1	1	47
resources	33	0	0	3	0	0	36
events	29	0	0	0	0	0	29
manifests	29	0	0	0	0	0	29
types	26	0	0	0	0	0	26
locations	24	1	0	0	0	0	25
instances	1	0	0	0	0	21	22
servers	3	0	0	0	17	0	20
clients	0	0	0	0	17	0	17
content	16	0	0	0	0	0	16
devices	8	0	0	8	0	0	16
organizations	16	0	0	0	0	0	16
services	16	0	0	0	0	0	16
views	16	0	0	0	0	0	16
vuln	16	0	0	0	0	0	16
shows	5	0	4	6	0	0	15
products	13	1	0	0	0	0	14
currencies	13	0	0	0	0	0	13
documents	13	0	0	0	0	0	13
files	7	0	0	6	0	0	13
jobs	2	10	0	0	0	1	13
artifacts	12	0	0	0	0	0	12
tasks	5	0	0	2	0	5	12
tags	8	0	3	0	0	0	11
images	9	1	0	0	0	1	11
keys	0	0	0	11	0	0	11
configs	2	8	0	0	0	0	10
descriptor	10	0	0	0	0	0	10
episodes	0	0	0	10	0	0	10
networks	3	0	0	6	0	1	10
people	4	0	0	6	0	0	10
versions	9	0	0	0	0	1	10
api-docs	9	0	0	0	0	0	9
email_history	9	0	0	0	0	0	9
groups	9	0	0	0	0	0	9
items	9	0	0	0	0	0	9
metadata	8	0	0	0	0	1	9
policydefinitions	3	0	0	6	0	0	9
roles	9	0	0	0	0	0	9
accounts	6	0	0	2	0	0	8
applications	8	0	0	0	0	0	8
concepts	8	0	0	0	0	0	8
config_schemas	8	0	0	0	0	0	8

Table 21: Enumerable Collection – Most occurring labels

D.2 Appendable Collection (P2)

user and *order* are the most used labels in this primitive. While most of the rest of the labels appear less than 10 times. GET/PUT/DEL (P2.v1) and GET/DEL (P2.v2) are very recurrent variants (occurrences respectively 194 and 145), however we notice that they are not used with a high number of distinct labels such as GET (P2.v3), which is the most occurring variant in this primitive.

	P2.v1	P2.v2	P2.v3	P2.s2	P2.s1	OCC
user	114	5	11	3	7	140
order	4	78	1	1	1	85
todo	1	12	1	0	0	14
messages	2	1	9	0	0	12
deployments	0	1	11	0	0	12
images	0	0	11	1	0	12
token	0	0	1	11	0	12
register	11	0	0	0	0	11
objectstores	0	0	0	0	8	8
account	1	0	6	0	0	7
comment	0	1	0	1	5	7
files	0	1	6	0	0	7
form_fields	0	0	6	0	0	6
labels	0	0	1	5	0	6
wall_comments	0	0	6	0	0	6
courses	1	4	0	0	0	5
disease	0	0	5	0	0	5
item	1	1	3	0	0	5
media	0	4	1	0	0	5
read	0	0	5	0	0	5
target	0	0	5	0	0	5
categorias	0	0	0	0	4	4
productos	0	0	0	0	4	4
post	0	0	2	1	1	4
products	2	0	2	0	0	4
traces	0	0	4	0	0	4
annotations	3	0	0	0	0	3
annotationsets	3	0	0	0	0	3
assets	0	0	0	3	0	3
cart	0	1	2	0	0	3
cluster	0	2	0	0	0	2
collections	0	0	3	0	0	3
datapointers	3	0	0	0	0	3
employee	0	1	1	0	1	3
entries	1	0	2	0	0	3
hub	0	2	0	1	0	3
individuals	3	0	0	0	0	3
jobs	0	0	1	2	0	3
policy_keys	0	0	3	0	0	3
provider	2	0	0	0	1	3
student	1	1	1	0	0	3
subscriptions	1	0	0	2	0	3
task	1	0	1	0	1	3
update-requests	0	3	0	0	0	3
wallet	0	0	2	0	1	3
category	1	0	0	0	0	2
balance	0	0	2	0	0	2
batch	0	0	2	0	0	2
book	0	0	1	0	0	1
bookings	0	1	0	1	0	2
buy	0	0	2	0	0	2
campaigns	0	2	0	0	0	2
client	1	0	1	0	0	2
connections	0	0	2	0	0	2
distributions	2	0	0	0	0	2
documents	0	2	0	0	0	2
experiments	0	0	2	0	0	2

Table 22: Appendable Collection – Most occurring labels

D.3 Collection (P3)

In this primitive, there is a dominant label which is *users*. It occurs 223 while the second most occurring labels, *audit_trails* and *change_logs* are only used 61 time.

	P3.v1	P3.v2	P3.v3	P3.v4	P3.v5	P3.v6	P3.v7	P3.s1	P3.s2	OCC
users	4	108	10	11	31	19	7	3	30	223
audit_trails	61	0	0	0	0	0	0	0	0	61
change_logs	61	0	0	0	0	0	0	0	0	61
projects	0	24	3	1	1	8	7	8	2	54
tags	1	4	3	0	26	0	12	2	1	49
articles	0	31	1	0	2	2	0	0	0	36
comments	0	8	1	1	1	3	18	0	2	34
products	3	23	4	0	1	1	0	1	1	34
order	0	11	1	2	1	4	2	2	8	31
rules	0	4	1	0	8	8	1	0	8	30
events	1	16	1	0	0	1	1	0	6	26
roles	1	13	2	2	1	3	0	1	3	26
clusters	1	2	12	0	0	3	0	7	0	25
policies	0	24	1	0	0	0	0	0	0	25
client	0	16	1	2	1	1	1	0	1	23
group	0	9	0	1	0	1	0	0	10	21
messages	0	2	4	0	0	5	4	0	6	21
posts	0	12	1	1	0	3	0	0	2	19
subscriptions	0	9	1	0	0	8	0	0	1	19
accounts	0	6	1	2	1	0	2	0	6	18
compositetypes	18	0	0	0	0	0	0	0	0	18
images	1	1	0	0	0	14	0	0	2	18
pets	0	1	1	0	0	4	0	0	12	18
tasks	2	3	4	1	1	2	0	0	5	18
applications	2	6	0	0	0	2	0	7	0	17
credentials	0	0	0	0	1	0	16	0	0	17
members	0	5	0	0	0	4	6	0	1	16
contacts	0	14	0	0	0	0	0	0	1	15
item	1	9	2	1	1	1	0	0	0	15
volumes	0	0	0	0	8	7	0	0	0	15
example_entities	3	11	0	0	0	0	0	0	0	14
services	1	10	0	0	0	2	1	0	0	14
types	8	1	1	0	1	1	0	1	1	14
authorizedcertificates	0	0	13	0	0	0	0	0	0	13
categories	1	8	1	0	0	2	0	1	0	13
domainmappings	0	0	13	0	0	0	0	0	0	13
ingressrules	0	0	13	0	0	0	0	0	0	13
invoices	2	9	0	0	0	1	0	0	1	13
sessions	0	0	4	0	0	7	0	0	2	13
bookings	0	3	1	0	2	0	6	0	0	12
books	0	9	1	0	0	2	0	0	0	12
configs	0	2	0	0	0	0	0	0	10	12
service-profiles	0	0	0	0	0	6	0	6	0	12
actions	0	8	3	0	0	0	0	0	0	11
apikeys	0	8	0	0	0	1	2	0	0	11
payments	1	5	3	0	0	0	0	0	2	11
tracks	0	8	0	0	1	0	1	0	1	11
cities	0	4	0	0	6	0	0	0	0	10
collaborators	0	0	0	0	0	10	0	0	0	10
networks	1	0	0	0	0	9	0	0	0	10
note	0	8	2	0	0	0	0	0	0	10
reward	0	0	0	0	0	0	0	0	10	10
rollouts	0	0	0	0	0	0	0	0	10	10
node	0	3	2	0	1	3	0	0	0	9
notifications	0	6	0	1	1	0	0	0	1	9
patient_health_metric	0	0	0	0	0	0	0	0	9	9
reward_earning	0	0	0	0	0	0	0	0	9	9
reward_earning_fulfillment	0	0	0	0	0	0	0	0	9	9
reward_program_activation	0	0	0	0	0	0	0	0	9	9
runs	0	1	2	0	0	0	6	0	0	9
webresource	9	0	0	0	0	0	0	0	0	9
actionalias	0	8	0	0	0	0	0	0	0	8
addresses	0	5	0	1	1	1	0	0	0	8
authors	0	7	1	0	0	0	0	0	0	8
calendar_event	0	0	8	0	0	0	0	0	0	8
clusterpairs	0	0	0	0	0	0	8	0	0	8
customers	0	5	0	0	1	0	0	0	2	8
device-profiles	0	0	0	0	0	4	0	4	0	8
environment	1	3	1	0	0	1	0	2	0	8
games	0	7	0	0	0	0	0	0	1	8
invitations	0	0	0	0	0	0	8	0	0	8
operations	0	0	0	4	0	0	1	0	3	8
profile	1	1	0	1	0	2	0	0	3	8

Table 23: Collection – Most occurring labels

D.4 Mutable Collection (P4)

This primitive is less frequently used. The intersection between the labels used with the "Remove All" variant and the two smells is rather small, resulting in the two labels: secrets and tags.

	P4.v1	P4.s1	P4.s2	OCC
roles	0	8	14	22
users	0	3	16	19
collaborators	0	0	13	13
schedulepolicies	0	8	0	8
storagepolicies	0	8	0	8
events	4	0	3	7
schedules	0	7	0	7
secrets	4	2	0	6
certificates	4	0	0	4
ingresses	4	0	0	4
fxorders	0	0	3	3
info	0	3	0	3
payments	0	0	3	3
workspaces	3	0	0	3
post	0	0	2	2
skill	0	0	2	2
appbindings	2	0	0	2
catalog	0	0	2	2
configmaps	2	0	0	2
employees	0	0	2	2
groups	0	0	2	2
kubedboperators	2	0	0	2
lists	0	0	2	2
messages	1	0	1	2
product	0	1	1	2
tags	1	1	0	2
virtualmachineinstancemigrations	2	0	0	2
virtualmachineinstancepresets	2	0	0	2
virtualmachineinstancereplicasets	2	0	0	2
virtualmachineinstances	2	0	0	2
virtualmachines	2	0	0	2
campaignrecipient	0	0	1	1
campaignsettings	0	0	1	1
cursus	0	0	1	1
listcampaigndefaults	0	0	1	1
listcontact	0	0	1	1
member	0	0	1	1
template	0	0	1	1
topic	0	0	1	1
word	0	0	1	1
accounts	0	0	1	1
aerospikeclusters	1	0	0	1
aerospikenamespacebackups	1	0	0	1
aerospikenamespacerestores	1	0	0	1
agent-class-manifest-config	0	0	1	1
agentpools	0	1	0	1
albs	1	0	0	1
apps	0	0	1	1
attacks	0	0	1	1
attributes	0	1	0	1
auditregistrations	1	0	0	1
beverages	0	1	0	1
blogs	0	0	1	1
boards	1	0	0	1
books	0	0	1	1
branches	1	0	0	1
bundles	1	0	0	1
calories	0	1	0	1
categories	0	0	1	1
cdiconfigs	1	0	0	1

Table 24: Mutable Collection – Most occurring labels

REFERENCES

[1] [n.d.]. OpenAPI Generator. https://github.com/OpenAPITools/openapi-generator. https://github.com/OpenAPITools/openapi-generator

[2] [n.d.]. Swagger Codegen. https://swagger.io/tools/swagger-codegen/. https://swagger.io/tools/swagger-codegen/

[3] Subbu Allamaraju. 2010. *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. " O'Reilly Media, Inc.".

[4] Önder Babur and Loek Cleophas. 2017. Using n-grams for the Automated Clustering of Structural Models. In *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 510–524.

[5] Robert Daigneau. 2011. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional, New York, NY, USA.

[6] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley, USA.

[7] Martin Fowler. 2010. Richardson Maturity Model: steps toward the glory of REST. https://www.martinfowler.com/articles/richardsonMaturityModel.html

[8] Florian Haupt, Frank Leymann, and Karolina Vukojevic-Haupt. 2018. API Governance Support through the Structural Analysis of REST APIs. *Comput. Sci.* 33, 3–4 (Aug. 2018), 291–303.

[9] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

[10] Stefan Kapferer and Olaf Zimmermann. 2020. Domain-Driven Service Design. In *Service-Oriented Computing*, Schahram Dustdar (Ed.). Springer International Publishing, Cham, 189–208.

[11] István Koren and Ralf Klamma. 2018. The exploitation of openapi documentation for the generation of web frontends. In *Companion Proceedings of the The Web Conference 2018*. 781–787.

[12] M. Maleshkova, C. Pedrinaci, and J. Domingue. 2010. Investigating Web APIs on the World Wide Web. In *2010 Eighth IEEE European Conference on Web Services*. 107–114. https://doi.org/10.1109/ECOWS.2010.9

[13] Francis Palma, Johann Dubois, Naouel Moha, and Yann-Gaël Guéhéneuc. 2014. Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach. In *Proc. of ICSOC*. Springer, 230–244.

[14] Sanjay Patni. 2017. *Pro RESTful APIs*. Springer.

[15] Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. 2016. A Pattern Language for RESTful Conversations. In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. Irsee, Germany.

[16] Fabio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. 2016. Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study. In *Proc. ICSOC*. Springer, 157–170.

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

[18] Leonard Richardson and Sam Ruby. 2007. *RESTful Web Services*. O'Reilly.

[19] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. 2016. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In *Proc. ICWE*. Springer, Lugano, Switzerland, 21–39.

[20] Spacy. [n.d.]. Models Documentation. https://spacy.io/models/en

[21] Phil Sturgeon. 2016. *Build APIs you won't hate*. LeanPub. https://leanpub.com/build-apis-you-wont-hate

[22] The Open API Initiative. [n.d.]. OAI. https://openapis.org. https://openapis.org/

[23] Markus Voelter, Michael Kircher, and Uwe Zdun. 2004. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. J. Wiley & Sons, Hoboken, NJ, USA.

[24] Jim Webber, Savas Parastatidis, and Ian Robinson. 2010. *REST in Practice: Hypermedia and Systems Architecture* (1st ed.). O'Reilly Media, Inc.

[25] Uwe Zdun and Paris Avgeriou. 2008. A catalog of architectural primitives for modeling architectural patterns. *Information and Software Technology* 50, 9 (2008), 1003–1034. https://doi.org/10.1016/j.infsof.2007.09.003

[26] Olaf Zimmermann, Daniel Lübke, Uwe Zdun, Cesare Pautasso, and Mirko Stocker. 2020. Interface Responsibility Patterns: Processing Resources and Operation Responsibilities. In *Proc. of the European Conference on Pattern Languages of Programs* (Online) *(EuroPLoP '20)*.

[27] Olaf Zimmermann, Daniel Pautasso, Cesare Lübke, Uwe Zdun, , and Mirko Stocker. 2019. Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources. In *Proc. of the European Conference on Pattern Languages of Programs* (Online) *(EuroPLoP '19)*.

[28] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2020. Introduction to Microservice API Patterns (MAP). In *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019) (OpenAccess Series in Informatics (OASIcs))*, Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh (Eds.), Vol. 78. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:17. https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4

[29] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2021. Microservice API Patterns. https://microservice-api-patterns.org/.

[30] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In *Proc. of the 22nd European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLoP '17)*. ACM, Article 27, 36 pages. https://doi.org/10.1145/3147704.3147734