Towards Extracting Web API Specifications from Documentation

Jinqiu Yang
University of Waterloo
Waterloo, Ontario, Canada
j223yang@uwaterloo.ca

Erik Wittern
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
witternj@us.ibm.com

Annie T.T. Ying
EquitySim
Vancouver, BC, Canada
annie.ying@gmail.com

Julian Dolby
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
dolby@us.ibm.com

Lin Tan
University of Waterloo
Waterloo, ON, Canada
lintan@uwaterloo.ca

ABSTRACT

Web API specifications are machine-readable descriptions of APIs. These specifications, in combination with related tooling, simplify and support the consumption of APIs. However, despite the increased distribution of web APIs, specifications are rare and their creation and maintenance heavily rely on manual efforts by third parties. In this paper, we propose an automatic approach and an associated tool called D2Spec for extracting significant parts of such specifications from web API documentation pages. Given a seed online documentation page of an API, D2Spec first crawls all documentation pages on the API, and then uses a set of machine-learning techniques to extract the base URL, path templates, and HTTP methods – collectively describing the endpoints of the API.

We evaluate whether D2Spec can accurately extract endpoints from documentation on 116 web APIs. The results show that D2Spec achieves a precision of 87.1% in identifying base URLs, a precision of 80.3% and a recall of 80.9% in generating path templates, and a precision of 83.8% and a recall of 77.2% in extracting HTTP methods. In addition, in an evaluation on 64 APIs with pre-existing API specifications, D2Spec revealed many inconsistencies between web API documentation and their corresponding publicly available specifications. API consumers would benefit from D2Spec pointing them to, and allowing them thus to fix, such inconsistencies.

ACM Reference Format:
Jinqiu Yang, Erik Wittern, Annie T.T. Ying, Julian Dolby, and Lin Tan. 2018. Towards Extracting Web API Specifications from Documentation. In *MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196398.3196411

1 INTRODUCTION

Web Application Programming Interfaces (web APIs or simply *APIs* from hereon) provide applications remote, programmatic access to resources such as data or functionalities. For application developers, the proliferation of such APIs provides tremendous opportunities. Applications can take advantage of vast amount of existing data, like location-based information from the Google Places API[1], hook into established and global social networks, using for example Facebook's[2] or Twitter's[3] APIs, or outsource critical and hard to implement functionalities, such as payment processing using the Stripe API.[4]

To consume APIs, though, developers face numerous challenges [30]: The need to find and select the APIs meeting their requirements, both from a functional and non-functional point-of-view [28]. They need to familiarize with the capabilities provided by an API and how to invoke these capabilities, which typically involves studying HTML-based documentation pages that vary across APIs. Compared to using library APIs, for example of a Java library, consumers of web APIs do not have interface signatures readily available or accessible via development tools. In addition, web APIs are under the control of independent providers who can change the API in a way that can break client code [13, 17]. Even for supposedly standardized notions such as the APIs' URL structures, HTTP methods, or HTTP status codes, the semantics and implementation styles differ across APIs [22].

One attempt to mitigate these problems is to describe APIs in a well-defined way using web API *specifications*.[5] Web API specification formats, like the OpenAPI Specification [5] or the RESTful API Modeling Language (RAML) [7], describe the URL templates, HTTP methods, headers, parameters, and input and output data required to interact with an API. Being machine-understandable, web API specifications are the basis for various tools that support API consumption: specification are input for generating consistent API documentation pages[6], they are used to catalog APIs[7], to auto-generate software development kits that wrap APIs in various languages[8], or even to statically check client code for possible errors [29].

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196411

[1] https://developers.google.com/places
[2] https://developers.facebook.com/docs/graph-api
[3] https://developer.twitter.com/en/docs/api-reference-index
[4] https://stripe.com/docs/api
[5] We acknowledge that the term *specification* sometimes means a much more comprehensive description of an application's or system's syntax *and* semantics. Even though web API specifications like OpenAPI Specification do not provide any semantics, we use the term here nonetheless due to its prevalence in the web API community.
[6] e.g., https://editor.swagger.io/ or https://github.com/Rebilly/ReDoc
[7] e.g., https://apiharmony-open.mybluemix.net/ or https://any-api.com
[8] e.g., https://swagger.io/swagger-codegen or https://apimatic.docs.apiary.io

Unfortunately, client developers cannot leverage advantages from such tooling unless web API specifications are available. In contrast to over 19 thousand APIs listed in ProgrammableWeb [6] (an authoritative directory of web APIs referred as "The Journal of the API Economy"[9]), fewer than one thousand specifications are publicly available on APIs.guru [4], the largest publicly available directory of web API specifications.[10] In APIs.guru, third parties provide and maintain specifications, relying either on manual efforts or API-specific scripts that translate otherwise made available information about an API into a known specification format like an OpenAPI Specification. The resulting efforts may explain the limited availability of specifications.

The goal of this research is to provide client developers or API catalogs access to a much larger number of specifications by extracting them from much more prevalent, semi-structured online documentation (typically in form of HTML pages). Many software engineering researchers have looked into a similar problem but in the traditional library API context, namely, identifying Java method signatures from API documentation [11, 21, 25]. These approaches share the assumption that method signatures or code elements being extracted are written in Java, adhering to the specific Java grammar and conventions. Approaches capable of extracting web API endpoint descriptions not only require an adjustment to the search pattern for web API endpoints, but also need to account for the two common but distinct styles of web API documentation: an example-based style (e.g., the GitHub API documentation as shown in Figure 1 uses an example-based style, where the base URL `https://api.github.com` and the path template `/users/{username}/orgs` are embedded in free-form text and a `curl` command) and a more structured, reference-based documentation style (e.g., the Instagram API, Figure 2).

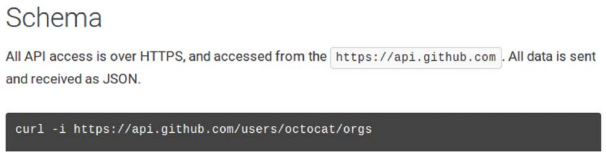


Figure 1: Example-style documentation (GitHub API)

In this paper, with these two distinct styles of web API documentation in mind, we propose an approach, and the associated tool D2Spec, to automatically extract a web API specification – more specifically, endpoint descriptions consisting of a base URL, path templates, and HTTP methods – from API documentation pages containing free-form text and arbitrary HTML structures. Given a seed documentation page for an API, D2Spec first crawls the associated documentation pages and then uses a set of machine learning techniques to extract the base URL of an API (e.g., `https://api.github.com`), the path templates (possibly containing path parameters, e.g., `/users/{username}/org`), and HTTP methods (e.g., `GET`, `POST`). More specifically, D2Spec uses classifiers and a hierarchical clustering algorithm to extract a base URL and path templates for an API, and searches the context of a path template to infer the HTTP method.

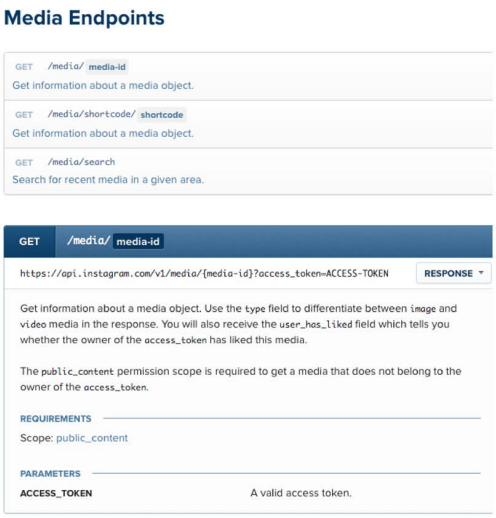


Figure 2: Reference-based API documentation (Instagram API)

We evaluated whether D2Spec can accurately extract endpoints from documentation on 116 web APIs. The results showed that D2Spec achieves a precision of 87.1% in identifying base URLs, a precision of 80.3% and a recall of 80.9% in generating path templates, and a precision of 83.8% and a recall of 77.2% in extracting HTTP methods. In addition, from an evaluation on 64 APIs with pre-existing API specifications, D2Spec revealed many inconsistencies between web API documentation and their corresponding publicly available specifications. API consumers could have benefited from D2Spec warning them about such inconsistencies, which may result in efforts to update specifications accordingly.

D2Spec currently does not infer the structure of data being sent to or received from an API, nor HTTP headers. However, since most requests use the `GET` method [22] and thus do not expect any request payload, in those cases, the combination of a base URL and path template already allow for a successful API invocation. Extending our work to extract the structure of data, for example through schema inference of provided example response data, is future work.

In the remainder of this paper, we present our approach to extract API specifications from documentation using a combination of machine learning techniques in Section 2. We present an empirical evaluation in Section 3. We discuss threats in Section 4 and related work in Section 5 before concluding in Section 6.

2 APPROACH

In this section, we describe how D2Spec combines machine learning classification and hierarchical clustering to extract significant parts of web API specifications from online documentation. D2Spec

[9] https://techcrunch.com/2013/08/07/veteran-news-editor-david-berlind-joins-programmable-web-the-journal-of-the-api-economy/

[10] While other catalogs like SwaggerHub.com may contain more specifications, these do to a large extend describe web APIs that are not publicly accessible.

focuses on extracting three components of API specifications: base URLs, path templates, and descriptions (i.e., HTTP methods).

The web API's **base URL** is essential in a web API specification: any URL of a Web API request must contain the base URL and the relative path of the corresponding endpoint. More formally, a base URL is a common prefix of all URLs for web API invocations, excluding other URLs such as documentation pages. In OpenAPI specifications, a base URL is constructed via three fields: a *scheme* (e.g., `https`), the *host* (e.g., `api.instagram.com`), and optionally a *base path* (e.g., `/v1`). In many APIs (e.g., *Instagram API*), the base URL is the *longest* common prefix of all the URLs for invoking the web API. However, for other APIs, such as Microsoft's *The DevTest Labs Client API*, the longest common prefix is `https://management. azure.com/subscriptions` while the actual base URL is `https://management.azure.com`, because `/subscriptions` is defined to be part of the endpoint paths. Whether a base URL is indeed the longest common prefix is a design decision of the API provider.

A **path template** defines fixed components of a URL as well as ones to be instantiated dynamically. For example, in the path template `/users/{userId}/posts`, the part `{userId}` is a *path parameter* that needs to be instantiated with a concrete value of a user ID before performing a request. A path parameter is typically denoted via enclosing brackets (i.e., "{}", "[]", "<>", or "()") or a prefix ":".

D2Spec focuses on one type of **description** associated with the path template: the **HTTP method**. It reflects the type of interaction to be performed on a resource exposed by a web API. While many web APIs long relied on `GET` and `POST`, now a much broader spectrum of methods is used [22]. As proposed in related work, we denote every valid combination of a path template and an HTTP method as an *endpoint* of the API [26].

D2Spec combines a set of techniques to infer the base URL, path templates, and HTTP methods, given a seed documentation page. Figure 3 provides an overview. In the first step, D2Spec uses a simple crawling approach to obtain complete documentation sources for an API. The crawler, starting from the provided seed page, iteratively downloads all linked *sub pages*. For crawling, D2Spec uses the headless browser Splash[11] to execute any JavaScript on each page before downloading it, as this may impact the resulting HTML structure. In order to extract the **base URL** (see Section 2.1), D2Spec first extracts all candidate URLs that can represent a web API call from the crawled documentation pages; D2Spec next leverages machine learning classification to determine for each candidate URL whether or not it is likely to represent an invocation to the documented web API. Finally, D2Spec selects the longest common prefix of these URLs. For the **path templates** (see Section 2.2), D2Spec leverages the URLs likely to be invocations of the API and extracts the URL paths (the part of the URL after removing the base URL). From these paths, using an agglomerative hierarchical clustering algorithm, D2Spec infers path templates by identifying path parameters and aggregating paths. D2Spec then finds the descriptions co-located with the URL paths in the documentation, from which it extracts the **HTTP method(s)** (see Section 2.3) that can be combined with each path template (forming endpoints).

[11] https://scrapinghub.com/splash/

2.1 Base URL Extraction

Identifying the base URL in online documentation is not as straightforward as searching for keywords or templates such as "The base URL is <*base URL*>"; base URLs are often not explicitly mentioned in the documentation. Rather, base URLs are often included as part of depicted examples of web API requests, as in the case of the GitHub documentation shown in Figure 1. Thus, D2Spec's approach is to infer the base URL from all the URLs provided in online documentation.

Step 1 - Extracting URLs

As a first step, D2Spec extracts all candidate URLs in the documentation that represent web API calls. These candidate URLs consist of standard URLs (according to the W3C definition [2]) and URLs containing path parameters enclosed in "`{}`", "`[]`", "`()`", or "`<>`". We do not include in this list URL links within `href` attributes of link tags, nor inside `<script>` tags: URLs that represent web API calls are one of the main content in a documentation page to be communicated to the readers; hence, such URLs tend to be rendered in the documentation rather than appear as links or in scripts. Even excluding such links, some of the URLs in the candidate list may not represent web API calls, e.g., URLs of related or even unrelated resources. In fact, we studied a set of 15 web APIs[12] and found that 42% of the contained URLs are *not* invocations of web APIs.

Step 2 - Identifying URLs of web API calls

To filter out spurious URLs that do not represent web API calls, we use supervised machine learning classification to determine whether each URL from Step 1 is likely to represent a web API call. For each URL, D2Spec generates two categories of features: properties of documentation pages and properties of the URL itself. The first category consists of four features related to properties of the documentation pages from which URLs were extracted:

- `clickable`: True, if the URL is part of the link text enclosed in `<a>` tags with the "`href`" attribute.
- `code_tag`: True, if the URL appears inside `<code>` tags, and false otherwise.
- `within_json`: True, if the URL is inside valid JSON within a pair of matched HTML tags.
- `same_domain_with_doc_link`: True, if the URL has the same host name as the URL of the documentation page itself, and false otherwise.

The second category consists of four features related to properties of the URL itself:

- `query_parameter`: True, if the URL contains query parameters which are denoted by ? and/or =. For example, in the URL `https://api.github.com/.../issues?state=closed`, `state` is a parameter with the value `closed`. URLs with parameters are more likely to depict web API calls.
- `api_convention`: The number of conventions exhibited by the URL indicates whether it likely corresponds to a web API call. We include three conventions described in previous work [22]: (1) whether the URL contains the term `rest`; (2) whether the URL contains the term `api`; and (3) whether the URL contains version related information, including the

[12] This data set of 15 web APIs, which we studied to design D2Spec, is independent of our evaluation set.

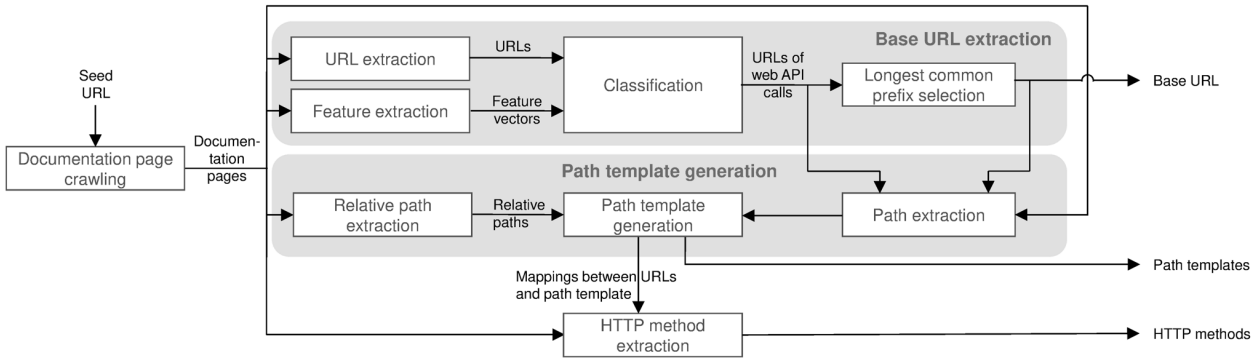


Figure 3: Overview of D2Spec, consisting of 'Base URL Generation' and 'Endpoint Generation'. D2Spec takes web API documentation as input and outputs specifications that include base URLs and endpoints.

terms `v[0-9\.]+` or `version[0-9\.]+`. For example, if a URL satisfies all three conventions, the value of this feature is 3.

- `path_template`: True, if the URL contains a path parameter denoted by enclosing brackets (`{}`, `[]`, `()`, `<>`) or a colon prefix (`:`).
- `curl_return`: A categorical feature representing the return value from invoking a `curl` command on the URL. We determine into which of the following categories the return value of the command falls: (1) is in JSON format (the URL likely corresponds to a web API request); (2) contains authentication errors[13] (the URL may correspond to a web request without the proper authentication); (3) everything else (e.g., in XML/HTML format which likely corresponds to learning resources as opposed to web API requests).

We built the classification model based on 15 web API documentation. This data set is independent of our evaluation data set. We manually identify a base URL for each web API in order to label whether each URL is indeed one web API call in the training set. Note that this manual process is only needed in order to build the training set; we do not need to manually identify base URLs when we apply D2Spec on web API documentation. In our evaluation, URLs are automatically labeled by machine learning classification. The feature vector of the URLs is automatically created by D2Spec.

D2Spec uses the support vector machine (SVM) classifier from scikit-learn [1] with the default parameters, both to train the model and to use the trained model in the evaluation as well. The trained model achieves an accuracy[14] of 0.97 and a F1-measure of 0.97 from the 10-fold cross-validation.

Step 3 - Extracting longest common prefix

Finally, D2Spec identifies the base URL by computing the longest common prefix of the URLs classified as likely depicting calls to the web API from step 2. This approach is based on the assumption that web API requests are the most frequent type of URLs rendered in the documentation. This assumption works well in practice, as the reported results in Section 3 show. This step is necessary because although the classification achieves high accuracy (97%), there are still URLs that do not target the web API.

2.2 Path Template Generation

Having identified an API's base URL as described in Section 2.1, we can use it to extract the path templates of the API.

Paths of endpoints are typically presented in a documentation page in one of two ways: Absolute URLs describe the whole URL used to perform an API request, for example, `https://api.github.com/repos/vmg/redcarpet/issues`. When identifying base URLs, D2Spec already extracts absolute URLs and can obtain paths of endpoints by truncating the already determined base URL. Alternatively, documentation pages may only provide relative path components without the base URL, for example `/users/repo`. In this case, D2Spec extracts relative paths based on heuristics (i.e., using regular expressions). In the experiment, we did not observe a significant number of false endpoint paths caused by this approach. From a manual analysis, we found that unlike URLs, which often include links to external resources, relative paths often describe API endpoints, since they are otherwise not very meaningful to a human.

In addition, path parameters are denoted in two ways: A path parameter can be denoted *explicitly* via enclosing syntactic constructs (e.g., "`{}`", "`[]`", "`<>`", or "`()`") or by prefixing a path parameter using "`:`". Other documentation pages *implicitly* indicate path parameters via an example style (e.g., the *GitHub* example in Figure 1), with URLs where parameters are instantiated. For example, in the URL `https://api.github.com/users/alice/gists`, "alice" is an instantiated value of the path parameter `{userId}`. Identifying path parameters expressed syntactically is straight-forward, while identifying path parameters in the example based documentation pages requires an algorithm that can determine from the path examples which of the path segments are instantiated values.

[13] We defined an authentication error to be indicated by a HTTP status code of either 401 or 407, or by a return message that contains the string "Invalid certificate".
[14] Accuracy is the percentage of correctly labeled (whether are indeed web API calls or not) URLs out of all the URLs extracted from the documentation.

Algorithm 1: Clustering algorithm

Input: *paths* /*a set of paths that represent endpoints*/
Input: T /* Threshold for merging clusters */
Output: $c_1, ..., c_n$ /*each cluster c_i groups the paths invoking the same endpoint*/

1 **Function** *hierarchical_clustering (T, paths)*
2 $C \leftarrow$ make each path a singleton cluster
3 **do**
4 $progress \leftarrow false$
5 **foreach** $c_i, c_j \in C$ *with min* $dist(c_i, c_j)$ **do**
6 **if** $dist(c_i, c_j) < T$ **then**
7 $progress \leftarrow true$
8 $C \leftarrow C - \{c_i, c_j\} \cup \{merge\ (c_i, c_j)\}$
9 **end**
10 **end**
11 **while** $|C| > 1 \wedge progress$;

We propose an iterative algorithm to infer whether a path segment is a fixed segment of an endpoint, a path parameter, or an instantiated value. The algorithm consists of two main ideas. First, it uses clustering to group paths that we infer to invoke the same endpoint. For example, if we found four paths in the documentation for an API:

```
/users/{username}/repos
/users/alice/repos
/users/alice/received_events
/users/bob/received_events
```

the clustering algorithm groups the first two into one cluster and the last two into the second cluster. From the first cluster, we know that `alice` is an instantiated value of `{username}`. Second, in subsequent iterations, the algorithm then leverages the fact that `alice` is an already inferred instantiated value to improve the clustering in the next iteration, marking both `alice` and `bob` as two instantiated values.

D2Spec uses hierarchical agglomerative clustering [18], as described in Algorithm 1. Given a set of paths with the same number of segments, the goal is to group the paths so that paths in a cluster invoke the same endpoint. We begin with one data point (i.e., one path) per cluster (line 2 in Algorithm 1). At each iteration (lines 4-10), we calculate the distance among all the pair-wise clusters and picks the pair with the shortest distance (line 5) to merge (line 8). For our implementation, the distance function (Algorithm 2) considers two paths the "closest" if they have exactly the same segments – each matching concrete (i.e. not a path parameter) segment i gets one point (Algorithm 2, line 8). Because two paths can never invoke the same endpoint when they have a different number of segments, the distance of such a pair is infinite (Algorithm 2, line 5). If the j-th segment of a path is a path parameter, the distance function considers the segment a match on the j-th segment of any other paths of the same length, with a discounted point of 0.8 instead of 1 (Algorithm 2, line 8). The clustering algorithm stops when the next pair of clusters to merge has the distance larger than a threshold T (Algorithm 1, lines 6, 7, and 11). In our implementation, the threshold is set to 1, meaning that we allow paths in a cluster to have a single path segment different from each other.

Algorithm 2: Distance functions

1 **Function** *dist(cluster* c_1*, cluster* c_2*)*
2 return $\min_{path_1 \in c_1, path_2 \in c_2} dist\_singles(path_1, path_2)$
3 **Function** *dist_singles(list of segments* S_1*, list of segments* S_2*)*
4 **if** $|S_1| \neq |S_2|$ **then**
5 return ∞
6 **end**
7 **else**
8 $sim \leftarrow \left(\begin{array}{l} |\{i \,|concrete(S_1[i]) \wedge S_1[i] = S_2[i]\}| + \\ 0.8 \times |\{i \,|param(S_1[j]) \vee param(S_2[j])\}| \end{array} \right)$
9 $d \leftarrow |S_1| - sim$
10 return d
11 **end**

To leverage the instantiated values that are already inferred, such as `alice` in the `received_events` cluster in the example, we adapt the standard hierarchical agglomerative algorithm as follows (Algorithm 3): The algorithm keeps track of a list of instantiated values of the path parameters per API (line 9), and stops when no additional instantiated values are found from the function *infer_parameter_value* (lines 10 and 12). Each iteration starts by updating the paths with the currently known instantiated values (lines 5-7). These paths are the input to the hierarchical agglomerative clustering algorithm (line 8). Clustering is performed after updating the newly instantiated values because when new path parameters are identified, the similarities will be updated. Within each cluster, new values of path parameters are inferred (line 10, the call to *infer_parameter_value*). This adapted algorithm can correctly cluster the four paths into two endpoints: `/users/{username}/repos` and `/users/ {username}/received_events`.

2.3 HTTP Methods

In web API documentation, the paths (whether or not path parameters are denoted explicitly or not) are typically co-located with other valuable information that D2Spec aims to extract, namely valid HTTP methods to use with a path template (`GET`, `PUT`, `DELETE`...). We call the context in which this information exists a *description block* of a path template. In this section, we first describe how we locate the description block associated with a path template, and then how D2Spec extracts the HTTP method.

D2Spec uses the URLs from the original documentation page that match with the inferred path templates as anchors in documentation pages (in HTML) to locate the scope of the description block. If there are multiple URLs in the page that match the path template, D2Spec combines the contexts of all the URLs as the description block of the path template. D2Spec locates a description block for each path template as follows. First, D2Spec parses the documentation page into a DOM tree (Figure 4), with each node representing the rendered text from the fragment of the HTML page enclosed in a pair of matched tags. Second, D2Spec marks the nodes whose rendered text contains at least one URL that matches a path template as gray, and locates the description block for each of these nodes. More specifically, for each gray node, D2Spec combines its description block by expanding to include (1) the siblings

Algorithm 3: Algorithm for inferring path parameters

```
Input: paths /*a set of paths that represent endpoints*/
Input: T /* Threshold for merging clusters */
Output: paths /*a set of paths with locations of path
        parameters identified*/
values ← ∅ /*the set of values of path parameters*/
do
    prevValueSize ← |values|
    foreach path ∈ paths do
        annotate the segments of path as parameters if they
        occur in values
    end
    clusters ← hierarchical_clustering(paths)
    foreach cluster ∈ clusters do
        values.addAll(infer_parameter_value(cluster))
    end
while prevValueSize ≠ |values|;
Function infer_parameter_value (cluster)
    paramValues ← ∅
    foreach pair (path, path_param) ∈ cluster do
        value ← extract the parameter value at the i-th
        segment in path where the i-th segment in
        path_param is a parameter paramValues.add(value)
    end
    return paramValues
```

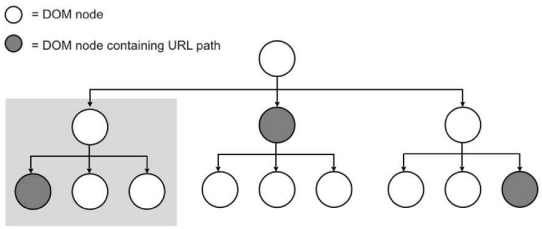


Figure 4: A DOM tree of an HTML page, with each node representing the rendered text from the fragment of the HTML page enclosed in a pair of matched tags. Gray nodes contain a path template. The gray box denotes an example *description block*.

of the node (starting with the closest siblings); then (2) the parent of the node. In (1), assuming the node is the n-th child of its parent, the expansion starts from $n-1$ to 0 and then $n+1$ to the farthest sibling; if a sibling is an ancestor of any other gray node, the expansion terminates entirely. This termination condition applies to 2) as well. The expansion stops if the parent is an ancestor of a node. An example description block is marked by the gray box in Figure 4.

In this work, D2Spec focuses on extracting HTTP methods for each path template.

Having determined description blocks, D2Spec searches in them for the seven possible method names, namely, `GET`, `POST`, `PUT`, `DELETE`, `OPTIONS`, `HEAD`, and `PATCH`. If none of these names is found, D2Spec uses `GET` as the default value for method of endpoint, as the GET method is the most popular method for web APIs [22].

3 EVALUATION

Client developers using web APIs can potentially benefit from D2Spec in two use cases: first, the tooling support made possible by the generated specifications, and second, warning of an inconsistency between a pre-existing specification and the API's documentation that may deserve further investigation. In this section, we present the evaluation of D2Spec. We aim to answer two questions, each addressing one of the use cases:

RQ1: *How accurately can D2Spec extract web API specifications from documentation?* We are interested in a quantitative perspective on how accurately D2Spec extracts base URL, path templates, and HTTP methods from the documentation.

RQ2: *Can D2Spec be used to identify inconsistencies between a pre-existing API specification and the API's documentation, pointing to the two being out of synchronization?* Some web APIs do have the associated specifications readily available. Third parties such as [4] (Section 3.1) curate specifications for many popular, public APIs. For other public or private APIs, the API providers themselves generate specifications. However, in either case, specifications can become out of synchronization with the documentation if, for example, a provider routinely maintains the one but not the other. We want to evaluate how well D2Spec can detect mismatches between documentation and corresponding specifications.

3.1 Data Collection

To address the two research questions, we need two types of information. For RQ1, we need API documentation pages to which we can apply D2Spec. We can then manually assess how accurate the resulting specifications are as compared to the documentation pages. To address RQ2, we need APIs that feature both documentation pages and existing specifications. After applying D2Spec to the documentation pages, we can assess whether the extracted specifications are in sync with the existing ones. We obtained these two types of information from two sources on March 22, 2017:

APIs.guru [4] is a third-party effort that curates OpenAPI specifications of popular public APIs. These specifications are either created and maintained through manual community effort, or using API-specific scripts, which translate other specification formats to the OpenAPI specification or are hard-coded to generate specifications from API-specific documentation pages. When we collected the evaluation data, APIs.guru hosted 276 specifications, each describing a different API with a unique base URL. Most of these APIs are provided by Google (127 APIs) or Microsoft (22 APIs), while the remaining 127 APIs from individual providers. We found documentation pages for all of the Google APIs, for none of the Microsoft APIs, and for 48 of the remaining APIs. If we were to construct the evaluation dataset for the machine learning model by including all APIs for which we found documentation pages, we risk unfairly evaluating the model that biases the Google APIs, whose documentation pages have the same HTML structure while the documentation pages from the other individual providers have different structures among them. Ultimately, we include 20 randomly

selected Google APIs in our evaluation. Overall, from APIs.guru, the APIs used in the evaluation consist of 48 APIs from the individual providers (we will call this set *GuruIndividual*) and 20 Google APIs (we will call this set *GuruGoogleSample*), thus a total of 68 APIs.

API Harmony [3] is a catalog of web APIs that helps developers to find and choose web APIs, and learn how to use them. API Harmony collects information on public web APIs. When we collected the evaluation data, API Harmony listed 1,019 web APIs in total, 772 of which contained links to the API's documentation page. We crawled the links to these documentation pages with the help of API Harmony's sitemap.xml file. We took a sample of 48 APIs (*HarmonySample*) from the 681 APIs from API Harmony and manually verified that the documentation indeed describes the API (from 772, we excluded 91 APIs that overlapped with APIs.guru).

From these two sources, collectively, we obtained 116 unique APIs. For RQ1 and RQ2, we will examine the set of 116 APIs as follows:

- The *GuruIndividual* dataset consists of the 48 APIs in APIs.guru that are not from Google or Microsoft.
- The *GuruGoogleSample* dataset consists of the 20 Google APIs from APIs.guru.
- The *HarmonySample* dataset consists of a sample of 48 APIs from the 681 APIs from API Harmony.

3.2 RQ1: Can D2Spec accurately extract web API specifications from documentation?

Approach: To assess the accuracy of D2Spec, we aim to determine how well the produced specifications match the input, which is the online documentation.

To increase the generalizability of our results, we performed the evaluation in two stages: First, we applied D2Spec to all 68 APIs obtained from APIs.guru (*GuruIndividual* and *GuruGoogleSample*). These APIs do not contain the 15 APIs we used to train D2Spec (see Section 2.1). We decided to use all APIs from APIs.guru in this evaluation because the required manual examination of them is also required to conduct the second research question, and there are limited APIs in APIs.guru to study. Second, we performed the evaluation on *HarmonySample*, which is a completely separate data source from APIs.guru. The results for these APIs thus better quantify how well D2Spec can potentially generalize to other API documentation pages. We limited the number of APIs in *HarmonySample* because the evaluation requires significant manual effort.

Overall, for RQ1, we considered 116 APIs (*GuruIndividual* + *GuruGoogleSample* + *HarmonySample*). To create the ground truth, we manually identified base URLs, path templates and HTTP methods from web API documentation. We then compared the ground truth with the specifications created by D2Spec for the same API. For base URLs, we calculated *precision*, which is the percentage the base URLs generated by D2Spec that are correct. Since each API documentation describes only one base URL, and by design D2Spec generates one base URL for each API documentation, *recall* is equal to *precision* for base URLs. For path templates and HTTP methods, we consider *precision* to be the percentage the results generated by D2Spec that are correct and *recall* to be the percentage of the given information type (e.g., path templates) in the documentation that D2Spec correctly generates. Because path templates and HTTP methods can only be extracted if a base URL was previously detected (see Sections 2.2 and 2.3), we focus on APIs for which D2Spec was able to do so in these parts of the evaluation.

Table 1: Precision and recall of D2Spec

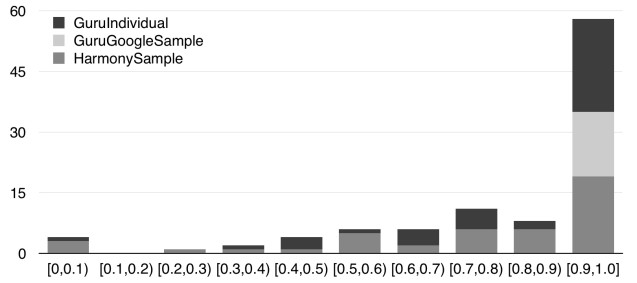
	HarmonySample (48 APIs)	*GuruGoogleSample* (20 APIs)	*GuruIndividual* (48 APIs)	All APIs (116 APIs)
Base URL				
# of APIs with correct base URL	45	16	40	101
Precision	93.8%	80.0%	83.3%	87.1%
Path Template				
# created D2Spec	967	188	1,331	2,486
# in documentation (with correct base URLs)	747	196	1,526	2,469
# matches	683	187	1,127	1,997
Precision	70.6%	99.5%	84.7%	80.3%
Recall	91.4%	95.4%	73.9%	80.9%
HTTP Method				
# created D2Spec	817	188	1,142	2,147
# in documentation (with correct base URLs)	815	219	1,297	2,331
# matches	658	184	957	1,799
Precision	80.5%	97.9%	83.8%	83.8%
Recall	80.7%	84.0%	73.8%	77.2%

Results: D2Spec recovered base URLs with a precision of 87.1%, inferred path templates with a precision of 80.3% and a recall of 80.9%, and extracted HTTP methods with a precision of 83.8% and a recall of 77.2%. Table 1 provides a break-down of the results.

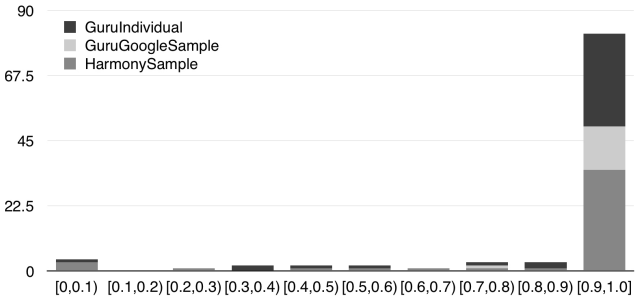
3.2.1 Base URL Results. For the 116 web APIs, D2Spec generated correct base URLs for 101 of them, yielding a precision of 87.1%. In the subsequent evaluation for path templates and HTTP methods for RQ1, the evaluation was based on the 101 APIs.

Upon manual inspection, we found that there were two reasons that D2Spec generated incorrect base URLs. First, when the documentation described multiple API versions, D2Spec was unable to tell which one was preferred by the writer of the documentation. For example, in the documentation of the *CityContext web API*, two endpoints were described with `https://api.citycontext.com/v1/postcodes` and `https://api.citycontext.com/v2/<location>`. D2Spec determined the base URL to be `https://api.citycontext.com` by selecting the longest common prefix of these two URLs. However, the official documentation listed `https://api.citycontext.com/v2` as base URL. Second, although the classification achieved a good precision, it is unable to remove all URLs that are not web API requests. Such URLs with the same prefix caused D2Spec to generate incorrect base URLs when they outnumbered the true web API requests.

3.2.2 Path Template Results. D2Spec was able to generate the majority (80.9% recall) of path templates correctly (80.3% precision) for the 101 web APIs whose base URLs are correctly identified by D2Spec. There were in total 2,469 path templates described in the documentation. D2Spec generated 2,486 path templates in total, and 1,997 of them were correct. Thus, the overall precision of path template extraction was 80.3%, and the recall was 80.9%. Figure 5 illustrates stacked histograms on precision and recall of the path



(a) Precision of D2Spec on the 101 web APIs.



(b) Recall of D2Spec on the 101 web APIs.

Figure 5: Stacked histograms showing precision and recall of D2Spec on the 101 web APIs for which the base URL was correctly extracted.

template extraction on the 101 APIs that D2Spec can generate correct base URLs for. For example, Figure 5a shows that for 58 (out of 101) web APIs, D2Spec achieves a precision above 90%.

3.2.3 HTTP Method Results. D2Spec achieved a precision of 83.8% and a recall of 77.2% in extracting HTTP methods for the path templates in the evaluated 101 web APIs. In total, there were 2,331 endpoints with the associated HTTP methods described in the web API documentation evaluated; D2Spec produced a result for 2,147 of them and 1,799 HTTP methods were correct. D2Spec failed to locate the correct HTTP method when its position in the documentation was far away from the path templates. For example, the *Mandrillapp* documentation has a consolidated description for all endpoints: "All API calls should be made with HTTP POST", instead of listing the method `POST` individually for each of the path template. Thus, D2Spec failed to identify correct method names for Mandrillapp's path templates.

3.3 RQ2: Can D2Spec be used to identify inconsistencies between a pre-existing API specification and the API's documentation, pointing to the two being out of synchronization?

Approach: We focused on the 68 APIs from APIs.guru (*GuruIndividual* + *GuruGoogleSample*). For these APIs, we compared the specifications generated by D2Spec (from hereon denoted as ***ToolSpecs***) with the specifications provided by APIs.guru (from hereon denoted as ***GuruSpecs***). Our comparison focused, again, on the three pieces of information extracted by D2Spec, namely, base URLs, path templates, and HTTP methods. For **base URLs**, we compared whether the ones extracted by D2Spec per API match those defined in the OpenAPI specifications. We obtained base URLs from the specifications by concatenating the `schemes`, `host`, and `basePath` fields. We then compared whether the extracted **path templates** and the associated **HTTP methods** match the ones in the specifications.

For each of the three extracted pieces of information, we counted the number of matches, and then manually inspected the mismatches to determine their origin.

Results: We found that mismatches between *GuruSpecs* and *ToolSpecs* were partly caused by limitations of D2Spec, and partly by publicly-available specifications (i.e., the *GuruSpecs*) being out of synchronization with API documentation: Our manual inspection showed that for base URLs and HTTP methods, *GuruSpecs* were often up-to-date with documentation, and mismatches between *ToolSpecs* and *GuruSpecs* were due to inaccuracies of D2Spec. However, for path templates, our manual inspection found that many mismatches were due to the documentation and *GuruSpecs* being out of synchronization with each other, or due to errors in the documentation. Specifically, for the 68 APIs evaluated, we identified 394 path templates from 24 APIs where *GuruSpecs* and the documentation were different. One reason for the mismatches is that as web APIs evolve, API providers tend to keep documentation up-to-date since it is, as a human-readable medium, often the first source that developers inquire to use APIs. In the following, we present the results of manually examining the mismatches between path templates in *GuruSpecs* and *ToolSpecs*. We found that mismatches fall into four categories:

- **Inconsistencies** were mismatches resulting from the documentation and specification in *GuruSpecs* being inconsistent with each other. Such inconsistencies were not caused by deficiencies of D2Spec or by errors in the documentation, but indicated that the API provider should either update the documentation or the specification. For example, in the documentation of *Slack*, there were eight endpoints on getting information on members from a given Slack team. The paths of all eight endpoints start with `/users.<action>`. However, only one path template was listed in the specification– `/users.list`; the remaining seven (e.g., `/users.info` and `/users.setPresence`) were missing in the specification. In this case, we considered that there were seven inconsistencies between the documentation and the specification.
- **Errors in the documentation** referred to obvious errors in the documentation, e.g., typos. Incorrect information in the documentation led D2Spec to generate path templates that, while being labeled as correct with regard to RQ1, did not match the specifications. For example, in the documentation of the *ClickMeter API*, many path templates starting with `/datatpoints` were misspelled as `/datapoints`. Thus, D2Spec generated several mismatched path templates compared to the official specification.
- **Partially correct path templates** occurred if D2Spec failed to infer path parameters correctly (i.e., the path templates

generated by D2Spec still contain path parameter values). A common reason for this problem was that the documentation contained only one instance of an instantiated value for a path parameter. In such cases, even though D2Spec's clustering-based algorithm can correctly place the path in its own cluster, D2Spec could not distinguish which segments of the path were instantiated values and which ones are fixed segments.

- **Deficiencies in the algorithm** occurred when D2Spec failed to extract certain path templates or generated incorrect path templates because of deficiencies in its design. For instance, D2Spec failed to extract certain path templates if the way the path templates appeared in the documentation was beyond the scope of the conventions used by D2Spec. D2Spec relies on the format of URL and relative path to extract path template information. If the path templates are not presented as such, D2Spec will not extract them correctly. For example, *HealthCare.gov*'s documentation describes a series of path templates as follows: "The following content types are available: articles, blog, questions, glossary, states, and topics. The request structure is https://.../api/:content-type.json." D2Spec extracts one path template–`/api/{content-type.json}` instead of six path templates (e.g., `/api/articles` and `/api/blog`). On the other hand, using the conventions mentioned above, D2Spec also extracted false path templates which did not describe path templates in the documentation. For example, the documentation of *dweet.io* listed a file path – `/play/definitions`, which was not a true path template.

Figure 6 visualizes the comparison of path templates from *GuruSpecs* and *ToolSpecs*. The breakdown of the mismatches from both aspects ($\neg$*ToolSpecs* $\cap$ *GuruSpecs* and *ToolSpecs* $\cap$ $\neg$*GuruSpecs*) is shown as well. In total, there are 929 path template matches between *ToolSpecs* and *GuruSpecs*. Among the 1509 path templates generated by D2Spec, there are 590 mismatches with the path templates defined in the *GuruSpecs*. Our manual analysis shows that 394 (67.8%) of the mismatches are caused by de-synchronization, i.e., "inconsistencies" and "errors in the doc.". The other two categories – "partially correct" and "deficiencies" – are due to the limitations of D2Spec.

Overall, while the manual examination of mismatches also pointed to some weaknesses of D2Spec, it also highlights that D2Spec can be used to find documentation and existing specifications being out of synchronization. To focus on this aspect, Figure 7 shows a histogram on the percentage of mismatches that are caused by de-synchronization for each web API. It shows that, for example, for 11 web APIs, over 90% of the mismatches detected by D2Spec indicate that documentation and pre-existing specifications were out of synchronization with each other.

4 THREATS TO VALIDITY AND DISCUSSIONS

Generalizability. D2Spec generates base URL from documentation by firstly identifying URLs that represent web API calls through a classification algorithm. The classification algorithm uses a set of pre-labeled URLs for training. We built the training set from a set of web API documentation that were independent of the ones used in the evaluation. The precision of D2Spec in base URL extraction

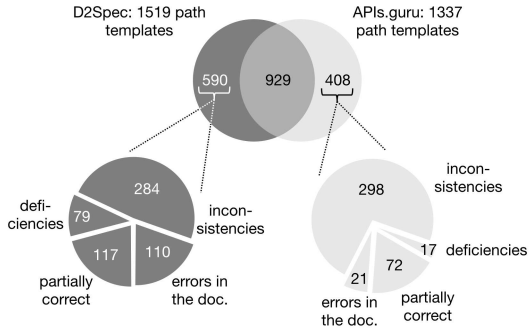


Figure 6: Comparison between path templates in specifications from D2Spec and the ones from APIs.guru.

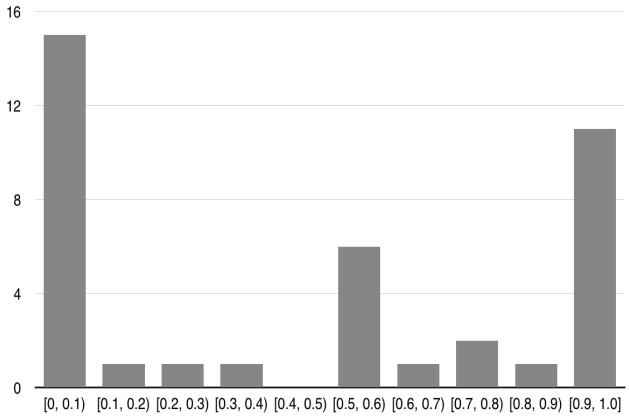


Figure 7: Percentage of "Inconsistencies" and "Errors on the documentation" between documentation and specifications by APIs.guru across APIs from APIs.guru

may be different if we use a different training set. However, we mitigated this bias by choosing a random set of web APIs for building the training set, and evaluating on a different set of APIs (the *GuruIndividual* and *GuruGoogleSample* datasets described in Section 3.1). In addition, we evaluated on a completely separate dataset (*HarmonySample*) and even achieved a better precision (97.5%) compared to a 80.0% precision on *GuruGoogle* and a 84.1% precision on *GuruIndividual*, demonstrating that our approach is likely to generalize to other unseen documentation pages.

Thresholds Used in the Clustering-Based Path Template Extraction. D2Spec leverages an iterative clustering-based algorithm to identify path parameters by inferring values of path parameters from *similar* web API calls. The proposed algorithm contains thresholds to control the hierarchical clustering (e.g., determining whether two web API calls are *similar* through a threshold T, see Algorithm 1 in Section 2.2). In this evaluation, we set the thresholds based on our observations on the training set. We found that the chosen thresholds also worked well for the evaluation set. Nevertheless, future studies should investigate the effects of different thresholds on the path template extraction results.

Documentation with Identical Structures. APIs from the same providers may have identical documentation structures (e.g., Google web APIs). Documentation structures may be different across different API providers. To show the generalizability of our approach, we applied our approach to APIs from different providers: Our evaluation set contains 120 APIs from 98 different web API providers.

5 RELATED WORK

We discuss related works that address extracting or inferring web API specifications, as well as works that rely on information extraction approaches, both for extracting software entities and for extracting any type of information from web pages generally.

Hanyang et al. describe *AutoREST*, a tool that, as does this work, aims to extract web API specifications from HTML-based documentation [9]. AutoREST uses a preprocessing step to select crawled web pages that likely contain information relevant to the specification, which could be used in combination with the here presented work. AutoREST relies on a set of simple, fixed rules to extract information from selected HTML pages, whereas the here presented methods are designed to be applicable also in light of stark differences in the way APIs are documented. We furthermore present a more extensive and detailed evaluation in this work. Gao et al. propose to infer constraints on the data required by web APIs (i.e., payload or parameters values) by mining both, API documentation and error-messages [15]. In contrast to the here presented work, the focus is thus on data definitions, making this work complementary to ours.

Further related works on extracting web API specifications rely on sources of information other than documentation. Wittern and Suter use dynamic traces in form of web-server logs [26]. The *SpyREST* tool, presented by Sohan et al., intercepts HTTP requests to an API using a proxy and then attempts to infer the API specification from them [23]. In later work, the same authors discuss the application of SpyREST at Cisco, where requests to the proxy are driven by existing tests against APIs [24]. Ed-douibi et al. propose an approach to generate web API specifications from example request-response pairs [12]. One benefit of our approach, as compared to these works, is that API documentation is typically publicly available, while access to web logs are limited to those with access to the private web servers, proxying may not be an option, and providing extensive examples for API usages may require (manual) effort, which could be targeted to generate specifications directly.

Many software engineering researchers have looked into the problem of identifying code elements—more specifically, Java code elements such as method signatures and calls—from API documentation. Dagenais and Robillard proposed an approach that extracts code elements from API documentation and links the elements to an index of known code elements, i.e., signatures from a Java library [11]. Subramanian et al. subsequently applied this approach to identify code elements on Stack Overflow posts and augmented the code elements in the posts with links to their official JavaDoc [25]. Rigby and Robillard use a light-weight, regular expression based approach to identify code elements that relaxes the requirement on a known index [21]. Another line of work focuses on extracting more complex specifications on code entities from natural language descriptions. Pandita et al. [20] extract method pre-conditions and post-conditions from natural language API documentation. Lin et al. [27] extract code contracts from comments and statically check for violations in the code. Our work differs in two ways. First, we extract web API endpoints and related information as opposed to code elements. Second, there is arguably greater value in our recovered index (i.e., OpenAPI Specifications) because such an index is often not available or known to the clients; while clients of Java libraries (or other statically-typed languages) are always exposed at least to method signatures, but callers of web APIs often do not have such information.

There have been many efforts in information extraction on web pages [8, 10, 14, 16, 19, 31, 32]. For example, techniques for extracting product information from e-commerce sites [31, 32] leverage the structure from the sites: the sites' organizational structure usually consists of a search page and a set of individual product pages, which typically have the same structure as they are generated from scripts. These techniques exploit this common structure across the pages within the same site. However, for extracting endpoints and other information from web API documentation pages, we cannot rely on such an assumption: There is no standard structure for API documentation. For many API documentation the content is semi-structured at best, written by humans using free-form text and/or diverse HTML structures. For example, the GitHub API documentation uses an example-based style, where the base URL `https://api.github.com` and the path template `/users/{username}/orgs` are embedded in free-form text and a `curl` command. Other documentation uses a more structured, reference-based documentation style.

6 CONCLUSION

In this paper, we presented D2Spec, a tool which extracts parts of web API specifications from documentation, including base URLs, path templates, and HTTP methods. D2Spec is based on the three assumptions: (1) documentation includes multiple web API URLs (so that a base URL can be extracted); (2) path templates are either denoted explicitly (e.g., using brackets) or that multiple example URLs for paths exist from which templates can be inferred; and (3) descriptions close to the path templates contain information about HTTP methods.

One missing piece so far is understanding the data that is returned by the APIs that we discover. We believe it is feasible to do this in several possible ways. The first is extending our extraction from documentation; documentation often includes example of API usage, and we could extract those examples and statically analyze that code for what data it expects back. Given example API usage, existing client code could be analyzed either dynamically or statically to infer data structures.

Our evaluation of D2Spec shows that our assumptions hold mostly true when it comes to extracting base URLs, path templates, and HTTP methods. It furthermore shows that D2Spec is not only useful for creating specifications from scratch, but also for checking existing ones for consistency with documentation. We contacted API providers for the found inconsistencies. In the future, we aim to expand the scope of D2Spec to also extract information on data structures, HTTP headers, and authentication methods.

REFERENCES

[1] 2016. scikit-learn. (2016). http://scikit-learn.org/stable/index.html.
[2] 2016. URL - W3C. (2016). https://www.w3.org/TR/url-1/.
[3] 2017. API Harmony. (2017). https://apiharmony-open.mybluemix.net.
[4] 2018. APIs.guru - Wikipedia for Web APIs. (2018). https://apis.guru/openapi-directory.
[5] 2018. OpenAPI Specification. (2018). https://github.com/OAI/OpenAPI-Specification.
[6] 2018. ProgrammableWeb API Directory. (2018). https://www.programmableweb.com/apis/directory/.
[7] 2018. RESTful API Modeling Language (RAML). (2018). https://raml.org/.
[8] Manuel Álvarez, Alberto Pan, Juan Raposo, Fernando Bellas, and Fidel Cacheda. 2008. Extracting lists of data records from semi-structured web pages. *Data & Knowledge Engineering* 64, 2 (2008), 491–509.
[9] Hanyang Cao, Jean-Rémy Falleri, and Xavier Blanc. 2017. Automated Generation of REST API Specification from Plain HTML Documentation. In *Service-Oriented Computing (ICSOC)*. Springer International Publishing, Cham, 453–461.
[10] Valter Crescenzi, Giansalvatore Mecca, Paolo Merialdo, et al. 2001. RoadRunner: Towards Automatic Data Extraction from Large Web Sites. In *Proceedings of the International Conference of Very Large Data Bases (VLDB)*. Morgan Kaufmann, 109–118.
[11] Barthélémy Dagenais and Martin P. Robillard. 2012. Recovering Traceability Links Between an API and Its Learning Resources. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE Press, 47–57.
[12] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2017. Example-Driven Web API Specification Discovery". In *Modelling Foundations and Applications*. Springer International Publishing, Cham, 267–284.
[13] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. 2015. Web API Growing Pains: Loosely Coupled yet Strongly Tied. *Journal of Systems and Software* 100 (2015), 27–43.
[14] Emilio Ferrara, Pasquale De Meo, Giacomo Fiumara, and Robert Baumgartner. 2014. Web data extraction, applications and techniques: a survey. *Knowledge-based systems* 70 (2014), 301–323.
[15] Chushu Gao, Jun Wei, Hua Zhong, and Tao Huang. 2014. Inferring Data Contract for Web-Based API. In *IEEE International Conference on Web Services (ICWS)*. IEEE, 65–72.
[16] Alberto HF Laender, Berthier A Ribeiro-Neto, Altigran S da Silva, and Juliana S Teixeira. 2002. A brief survey of web data extraction tools. *ACM Sigmod Record* 31, 2 (2002), 84–93.
[17] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. 2013. How Does Web Service API Evolution Affect Clients?. In *2013 IEEE 20th International Conference on Web Services (ICWS)*. 300–307.
[18] Manning. 2009. *Introduction to Information Retrieval*. Cambridge Press.
[19] Jussi Myllymaki. 2002. Effective web data extraction with standard XML technologies. *Computer Networks* 39, 5 (2002), 635–644.
[20] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring Method Specifications from Natural Language API Descriptions. In *Proceedings of the 34th International Conference on Software Engineering*. 815–825.
[21] Peter C Rigby and Martin P Robillard. 2013. Discovering essential code elements in informal documentation. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE Press, 832–841.
[22] 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 *Web Engineering*. Springer International Publishing, Cham, 21–39.
[23] S M Sohan, Craig Anslow, and Frank Maurer. 2015. SpyREST: Automated RESTful API Documentation Using an HTTP Proxy Server. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 271–276.
[24] S. M. Sohan, C. Anslow, and F. Maurer. 2017. Automated example oriented REST API documentation at Cisco. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 213–222.
[25] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 643–652.
[26] Philippe Suter and Erik Wittern. 2015. Inferring web API descriptions from usage data. In *Proceedings of the Third IEEE Workshop on Hot Topics in Web Systems and Technologies*. 7–12.
[27] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*Icomment: Bugs or Bad Comments?*/. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 145–158. https://doi.org/10.1145/1323293.1294276
[28] Erik Wittern, Vinod Muthusamy, Jim Alain Laredo, Maja Vukovic, Aleksander A. Slominski, Shriram Rajagopalan, Hani Jamjoom, and Arjun Natarajan. 2016. API Harmony: Graph-based search and selection of APIs in the cloud. *IBM Journal of Research and Development* 60, 2-3 (March 2016), 12:1–12:11.
[29] Erik Wittern, Annie T. T. Ying, Yunhui Zheng, Julian Dolby, and Jim Alain Laredo. 2017. Statically Checking Web API Requests in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE Press, 244–254.
[30] Erik Wittern, Annie T. T. Ying, Yunhui Zheng, Jim Alain Laredo, Julian Dolby, Christopher C. Young, and Aleksaner A. Slominski. 2017. Opportunities in Software Engineering Research for Web API Consumption. In *2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI)*. IEEE, 7–10.
[31] Yanhong Zhai and Bing Liu. 2005. Web data extraction based on partial tree alignment. In *Proceedings of the International Conference on World Wide Web*. ACM, 76–85.
[32] Yanhong Zhai and Bing Liu. 2007. Extracting web data using instance-based learning. *World Wide Web: Internet and Web Information Systems* 10, 2 (2007), 113–132.