Automatic Canonical Utterance Generation for Task-Oriented Bots from API Specifications

Mohammad-Ali Yaghoub-Zadeh-Fard
m.yaghoubzadehfard@unsw.edu.au
University of New South Wales, Sydney
Sydney, NSW

Boualem Benatallah
b.benatallah@cse.unsw.edu.au
University of New South Wales, Sydney
Sydney, NSW

Shayan Zamanirad
shayanz@cse.unsw.edu.au
University of New South Wales, Sydney
Sydney, NSW

ABSTRACT

With the mind-blowing development of REST (REpresentational State Transfer) APIs (Application Programming Interfaces), many applications have been designed to harness their potential. As such, bots have recently become interesting interfaces to connect humans to APIs. Supervised approaches for building bots rely upon a large set of user utterances paired with API methods. Collecting such pairs is typically done by obtaining initial utterances for a given API method and paraphrasing them to obtain new variations. However, existing approaches for generating initial utterances (e.g., creating sentence templates) do not scale and are domain-specific, making bots expensive to maintain. The automatic generation of initial utterances can be considered as a supervised translation task in which an API method is translated into an utterance. However, the key challenge is the lack of training data for training domain-independent models. In this paper, we propose *API2CAN*, a dataset containing 14,370 pairs of API methods and utterances. The dataset is built by processing a large number of public APIs. However, deep-learning-based approaches such as sequence-to-sequence models require larger sets of training samples (ideally millions of samples). To mitigate the absence of such large datasets, we formalize and define resources in REST APIs, and we propose a delexicalization technique (by converting an API method and initial utterances to tagged sequences of resources) to let deep-learning-based approaches learn from such datasets.

1 INTRODUCTION

Much of the information we receive about the world is API-regulated. Essentially, APIs are used for connecting devices, managing data, and invoking services [1–3]. In particular, because of its simplicity, REST is the most dominant approach for designing Web APIs [4–6]. Meanwhile, thanks to the advances in machine learning and availability of web services, building natural language interfaces has gained attention by both researchers and organizations (e.g., Apple's Siri, Google's Virtual Assistant, IBM's Watson, Microsoft's Cortana). Natural language interfaces and virtual assistants serve a wide range of tasks by mapping user utterances (also called user expressions) into appropriate operations. Examples include reporting weather, booking flights, controlling home devices, and querying databases [1, 7–9]. Increasingly, organizations have started or plan to use capabilities arising from advances in cognitive computing to increase productivity, automate business processes, and extend the breadth of their business offering.

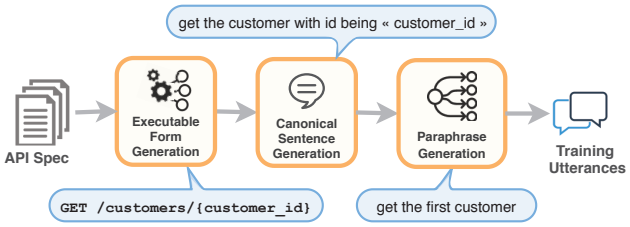
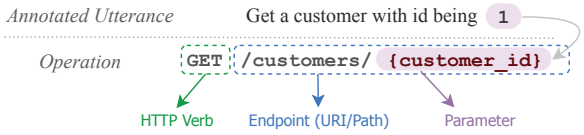


Figure 1: Classical Training Data Generation Pipeline

To serve users' requests, virtual assistants often employ supervised models which require a large set of natural language utterances (e.g., *"get a customer with id being 1"*) paired with their corresponding executable forms (e.g., SQL queries, API calls, logical forms). The training pairs are used to learn the mappings between user utterances and executable forms. Given the popularity of REST APIs (based on the well-known HTTP protocol), we focus on one of the most common types of executable forms called *operations*. In REST APIs, an operation (also called API method) consists of an HTTP verb (e.g., GET, POST), an endpoint (e.g., `/customers`), and a set of parameters[1] (e.g., query parameters). Figure 2 shows different parts of a REST request in HTTP.

An annotated utterance is a corresponding natural language expression to an operation in which API parameters are labeled:



As shown in Figure 1, collecting such pairs is typically done in two steps: (i) obtaining initial utterances for each operation; and (ii) paraphrasing the initial utterances either automatically or manually (e.g., crowdsourcing) to new variations in order to live up to the richness in human languages [1, 7, 8]. Paraphrasing approaches (e.g., crowdsourcing, automatic paraphrasing systems) have made the second step less costly [7, 8, 10], but existing approaches for generating the initial sentences are still limited, and they are not scalable [8].

Existing solutions for generating initial utterances (also called canonical utterances) often involve employing domain experts to generate hand-crafted domain-specific grammars or templates [1, 8, 11]. Almond virtual assistant, as an example, relies on hand-crafted rules to generate initial utterances [8]. Such approaches

[1] In this paper, to show parameters of an operation, we use curly brackets with two parts separated by semicolon (e.g., {customer_id:1}): the first part gives the name of the parameter and the second part indicates a sample value for the parameter

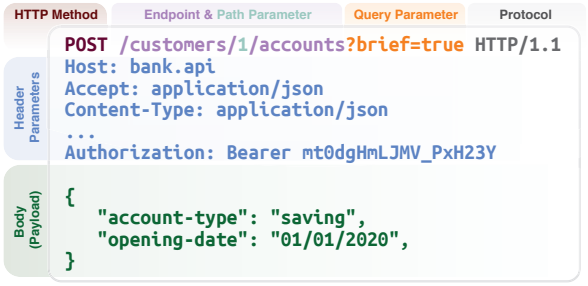


Figure 2: Example of an HTTP POST Request

are domain-specific and costly since rules are generated by experts [1, 8, 11]. In other words, adding new APIs to a particular virtual assistant requires manual efforts for revising hand-crafted grammars to generate training samples for new domains. With the growing number of APIs and modifications of existing APIs, automated bot development has become paramount, especially for virtual assistants which aim at servicing a wide range of tasks [1, 8].

Supervised approaches such as sequence-to-sequence models can be used for translating *operations* to *canonical utterances*. However, the key challenge is the lack of training data (pairs of operations and canonical utterances) for training domain-independent models. In this paper, we propose *API2CAN*, a dataset containing 14,370 pairs of operations and canonical utterances. The dataset is generated automatically by processing a large set of OpenAPI specifications[2] (based on the description/summary of each operation). However, deep-learning-based approaches such as sequence-to-sequence models require much larger sets of samples to train from (ideally millions of training samples). That is to say, sequence-to-sequence models are easy to overfit small training datasets, and issues such as out of vocabulary words (OOV) can negatively impact their performance. To overcome such issues, we propose a delexicalization technique to convert an operation to a sequence of predefined tags (e.g., singleton, collection) based on RESTful principles and design guidelines (e.g., use of plural names for a collection of resources, using HTTP verbs). In summary, our contribution is three-folded:

- **A Dataset.** We propose a dataset called *API2CAN*, containing annotated canonical templates (a canonical utterance in which parameter values have been replaced with placeholders e.g., *"get a customer with id being «id»"*) for 14,370 operations of 985 REST APIs. We automatically built the dataset by processing a large set of OpenAPI specifications, and we converted operation descriptions to canonical templates based on a set of heuristics (e.g., extracting a candidate sentence, injecting parameter placeholders in the method descriptions, removing unnecessary words). We then split the dataset into three parts (test, train, and validation sets).
- **A Delexicalization Technique.** Deep-learning algorithms such as sequence-to-sequence models require millions of training pairs to learn from. To assist such models to learn from smaller datasets, we propose a delexicalization technique to convert input (operation) and output (canonical template) of such models to a sequence of predefined tags called *resource identifiers*. The proposed approach is based on the concept of *resource* in RESTful design. Particularly, we formalize various kinds of resources (e.g., collection, singleton) in REST APIs. Next, using the identified resource types, we propose a delexicalization technique to replace mentions of each *resource* (e.g., customers) with a corresponding resource identifier (e.g., Collection_1). As such, for a given operation (e.g., `GET /customers/{customer_id}`), the model learns to translate the delexicalized operation (e.g., `GET Collection_1 Singleton_1`) to a delexicalized canonical templates (e.g., "get a Collection_1 with Singleton_1 being «Singleton_1»"). A resource identifier consists of two parts: (1) the type of resource and (2) a number n which indicates n-th occurrence of a resource type in a given operation. Resource identifiers are then used in time of translation to lexicalize the output of the sequence-to-sequence model (e.g., "get a Collection_1 with Singleton_1 being «Singleton_1»") to generate a canonical template (e.g., "get a customer with customer id being «customer_id»"). Delexicalization is done to reduce the impact of OOV and force the model to learn the pattern of translating resources in an operation to a canonical template (rather than translating a sequence of words).
- **Analysis of Public REST APIs.** We analyze and give insight into a large set of public REST APIs. It includes how REST APIs are designed in practice and drifts from the RESTful principles (design guidelines such as using plural names, appropriate use of HTTP verbs). We also provide inside into distribution of parameters (e.g., parameter types and location) and how values can be sampled various types of parameters to generate canonical utterances out of canonical templates using API specifications (e.g., example values, similar parameters with sample values). Automatic sampling values for parameters is essential for automatic generation of canonical utterances because current bot development platforms (e.g., IBM Watson) require annotated utterances (not canonical templates with placeholders).

2 RELATED WORK

REST APIs. REST is an architectural style and a guideline of how to use the HTTP protocol[3] for designing Web services [12]. RESTful web services leverage HTTP using specific architectural principles (i,e., addressability, uniform interface) [13]. Since REST is just a guideline without standardization, it is not surprising that API developers only partially follow the guidelines or interpret REST in their own ways [5]. In particular, this paper is built upon one of the most important principles in REST, namely the *uniform interface* principle. According to this principle, resources must be accessed and manipulated using proper HTTP methods (e.g., DELETE, GET) and status codes (e.g. using "201" to show a resource is created, and "404" to show resource does not exist). The uniform interface requires API to be developed uniformly to ensure that API users can understand the functionality of each operation without reading tedious and long descriptions. To ensure uniform interface, API developers are required to follow design patterns (e.g., using plural names to name collection of resources, using lowercase letters in paths). Existing works have listed not only those patterns but also anti-patterns in designing interfaces of REST APIs [5, 14, 15]. Examples of anti-patterns

[2] previously known as Swagger specification

[3] REST isn't protocol-specific, but it is designed over HTTP nowadays

also include using underline in paths and adding file extensions in paths [4, 6].

In this paper, we build upon existing works on designing interfaces for REST APIs. In particular, we formalize resource types based on patterns and anti-patterns recognized in prior works and built a resource tagger to annotate the segments of a given operation with resource types.

Conversational Agents and Web APIs. Research on conversational agents (e.i., bots, chatbots, dialog systems, virtual assistants) dates back to decades ago [16]. However, there have been only a few targeting web APIs, particularly because of the lack of training samples [1–3]. In absence of training data, operations descriptions (e.g., having long descriptions containing unnecessary information) have been used for detecting the user's intent [3]. However, operations often lack proper descriptions, and operations descriptions may share the same vocabularies in a single API, making it difficult for the bot to differentiate between operations [3]. Moreover, these descriptions are rarely similar to the natural language utterances which are used by bot users to interact with bots. That is to say, these descriptions are originally written to document operations (not intended to be used for training bots) [2, 3].

Other approaches rely on domain experts for generating initial utterances [1, 7, 8]. These approaches include (i) natural language templates (a canonical utterance with placeholders) which are written by experts [17], and (ii) domain-specific grammars such as rules written for semantic parsers [1, 8]. Thus in either approach, manual effort is required to modify the templates of grammar if API specifications are changed. In the template-based approach, for each operation, a few templates are created in which entities are replaced with placeholders (e.g., *"search for a flight from* `ORIGIN` *to* `DESTINATION`"). Next, by feeding values (e.g., `ORIGIN=[Sydney]` and `DESTINATION=[Houston]`) to the placeholders canonical utterances are generated (e.g., *"search for a flight from Sydney to Houston"*). Likewise, generative grammars have been used by semantic parsers for generating canonical utterances [1, 17, 18]. In this approach, logical forms are automatically generated based on the expert-written grammar rules. The grammar is used to automatically produce canonical utterances for the randomly generated logical forms [1]. Both generative grammar and template-based approaches require human efforts, making them hard and costly to scale.

In our work, by adopting ideas from the principles of RESTful design and machine translation techniques, we tackle the main issue which is creating the canonical utterances for RESTful APIs. As opposed to current techniques such as generative-grammar-based or template-based approaches, the proposed approach is domain-independent and can automatically generate initial utterances without human efforts. We thus pave the way for automating the process of building virtual assistants, which serve a large number of tasks, by automating the process of training datasets for new/updated APIs.

User Utterance Acquisition Methods. Current approaches for obtaining training utterances usually involves three main paradigms: launching a prototype to get utterances from end-users, employing crowd workers, and using automatic paraphrasing techniques to paraphrase existing utterances [19].

In the prototype-based approach, a bot is built without any (rule-based methods) or with a small number of annotated utterances. Such prototypes are able to obtain utterances from users to further improve the bots based on supervised machine learning techniques [20]. However, in case of using supervised machine learning methods in building the prototype, collecting initial annotated user utterances is still needed. Collecting an initial set of training samples is essential since the prototype bot must be accurate enough to serve existing user's requests without turning them away from the bot.

```
paths:
   /customers/{customer_id}:
     get:
        description: gets a customer by its id,
        summary: returns a customer by its id,
        parameters:
        - {
            name: customer_id,
            in: path,
            description: customer identifier,
            required: true,
            type: string
          }
```

Figure 3: Excerpt of an OpenAPI Specification

Crowdsourcing has been also used extensively to obtain natural language corpora for conversational agents [1, 8, 17, 18]. In this approach, a canonical utterance is provided as a starting point, and workers are asked to paraphrase the expression to new variations. Automatic paraphrasing techniques have also been employed to automatically generate training data [21–24]. This is done by paraphrasing canonical utterances to obtain new utterances automatically. However, while automatic paraphrasing is scalable and potentially cheaper, even the state-of-art models fall short in producing sufficiently diverse paraphrasing [25], and fail in producing multiple semantically-correct paraphrases for a single expression [26–28]. Nevertheless, these automatic approaches are still beneficial for bootstrapping a bot.

In this paper, we propose a dataset and an automated, scalable, and domain-independent approach for generating canonical utterances. Generated canonical utterances can be next fed to either automatic paraphrasing systems or crowdsourcing techniques to generate training samples for task-oriented bots.

3 THE API2CAN DATASET

In this section, we explain the process of building the API2CAN dataset, and we provide its statistics (e.g., size).

3.1 API2CAN Generation Process

To generate the training dataset (pairs of operations and canonical utterances), we obtained OpenAPI specifications indexed in OpenAPI Directory[4]. OpenAPI Directory is a Wikipedia for REST APIs[5], and OpenAPI specification is a standard documentation format for REST APIs. As shown in Figure 3, the OpenAPI specification includes description, and information about the parameters (e.g., data types, examples) of each operation. We obtained the latest version of each API index in OpenAPI Directory, and totally collected 983 APIs, containing 18,277 operations in total (18.59 operation per an API on average). Finally, we generated canonical utterances for each of the extracted operations as described in the rest of this section and illustrated in Figure 4.

Candidate Sentence Extraction. We extract a candidate sentence from either the summary or description of the operation specification. For a given operation, the description (and summary) of the operation (e.g., *"gets a [customer] (#/definitions/-Customer) by id. The response contains ..."*) is pre-processed by

[4]https://github.com/APIs-guru/openapi-directory/tree/master/APIs
[5]https://apis.guru/browse-apis/

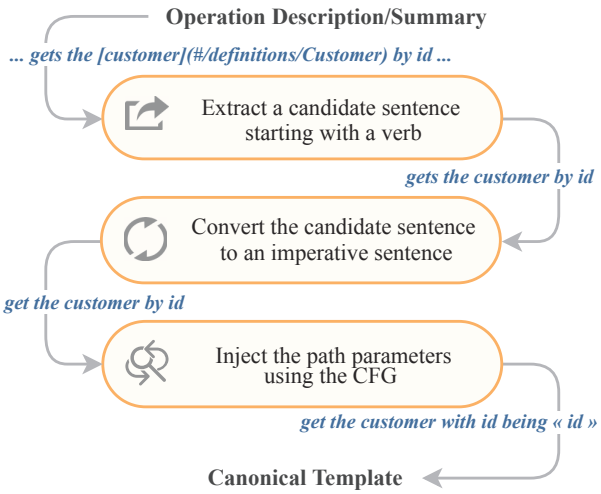


Figure 4: Process of Canonical Utterance Extraction

removing HTML tags, lowercasing, and removing hyperlinks (e.g., *"gets a customer by id. the response contains ..."*) and then it is split into its sentences (e.g., "gets a customer by id.", "the response contains ..."). Next, the first sentence starting with a verb (e.g., *"gets a customer by id"*) is chosen as a potential canonical utterance, and its verb is converted to its imperative form (e.g., *"get a customer by id"*).

Parameter Injection While the extracted sentence is usually a proper English sentence, it cannot be considered as a user utterance. That is because the sentence often points to the parameters of the operation without specifying their values. For example, given an operation like "`GET /customers/{customer_id}`" the extracted sentence is often similar to sentences like *"get a customer by id"* or *"return a customer"*. However, we are interested in annotated canonical utterances such as *"get the customer with id being «id»"*, and *"get the customer when its id is «id»"*; where "«id»" is a sampled valued for `customer_id`. To consider parameter values in the extracted sentence, we created a context-free grammar (CFG) as briefly shown in Table 1. This grammar has been created based on our observations of how operation descriptions are written (how parameters are mentioned in the extracted candidate sentences) by API developers. With this grammar, a list of possible mentions of parameters in the operation description is generated (e.g., *"by customer id"*, *"based on id"*, *"with the specified id"*). Then the lengthiest mention found in the sentence is replaced with *"with NPN being «PN»"*, where *NPN* and *PN* are human-readable version of the parameter name (e.g., customer_id ⟶ customer id) and its actual name respectively (e.g., *"get a customer with customer id being «customer_id»"*).

We also observed that path parameters are not usually mentioned in operation descriptions in API specifications. For example, in an operation description like *"returns an account for a given customer"* the path parameter *accountId* and *customerId* are absent, but the lemmatized name of collections *"customer"* and *"account"* are present. By using the information obtained from detecting such resources (see Section 4.2), it is possible to convert the description into *"return an account with id being «customer_id» for a given customer with id being «account_id»"*.

In the process of generating the *API2CAN* dataset, a few types of parameters were automatically ignored. As such, we did not include *header parameters*[6] since they are mostly used for authentication, caching, or exchanging information such as *Content-Type* and *User-Agent*. Thus such parameters do not specify entities of users' intentions. Likewise, using a list of predefined parameter names (e.g., auth, v1.1), we automatically ignored *authentication* and *versioning parameters* because bot users are not expected to directly specify such parameters while talking to a bot. Moreover, since the payload of an operation can contain inner objects, we assume that all attributes in the expected payload of an operation are flattened. This is done by concatenating the ancestors' attributes with the inner objects' attributes. For instance, the parameters in the following payload are flattened to "customer name" and "customer surname":

```
{
  "customer": {
    "name": "string",
    "surname": "string"
  }
}
```

As such, we convert complex objects to a list of parameters that can be asked from a user during a conversation.

Table 1: Parameter Replacement Context Free Grammar

Rule	
$N \longrightarrow \{PN\}\|\{NPN\}\|\{LPN\}\|\{RN\}\|\{NRN\}\|\{LRN\}$	
$CPX \longrightarrow$ 'by' | 'based on' | 'by given' | 'based on given' | ...	
$R \longrightarrow N \mid CPX\ N \mid N\ \text{CPX}\ N$	
{PN}	Parameter Name (e.g., "customer_id", "CustomerID", "CustomersID")
{NPN}	Normalized PN by splitting concatenated words and lowercasing (e.g., "customer id", "customers id")
{LPN}	Lemmatized NPN (e.g., "customer id")
{RN}	Resource Name (e.g., "Customers")
{NRN}	Normalized RN (e.g., "customers")
{RN}	Lemmatized NRN (e.g., "customer")

3.2 Dataset Statistics

By processing all API specifications, we were able to automatically generate a dataset called *API2CAN*[7] which includes 14,370 pairs of operations and their corresponding canonical utterances. We next divided the dataset into three parts as summarized in Table 2, and manually checked and corrected extracted utterances in the test dataset to ensure a fair assessment of models learned on the dataset[8].

Table 2: *API2CAN* Statistics

Dataset	APIs	Size
Train Dataset	858	13029
Validation Dataset	50	433
Test Dataset	50	908

Figure 5 shows the number of operations in *API2CAN* based on the HTTP verbs (e.g., GET, POST). As shown in Figure 5, the

[6] Header fields are components of the header section of request in the Hypertext Transfer Protocol (HTTP).
[7] https://github.com/mysilver/API2CAN
[8] Train and validation datasets will be also manually revised in near future

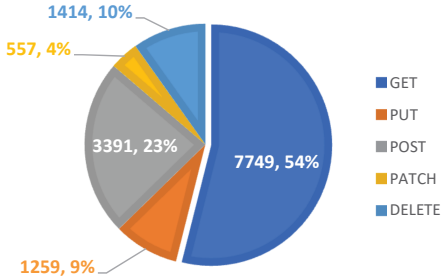


Figure 5: API2CAN Breakdown by HTTP Verb

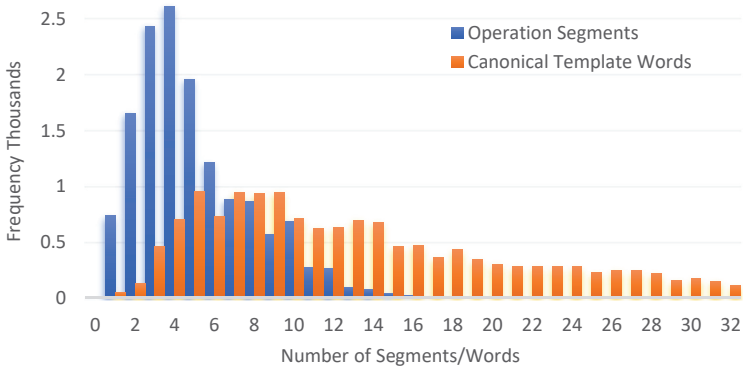


Figure 6: API2CAN Breakdown by Length

majority of operations are of GET methods which are usually used for retrieving information (e.g., *"get the list of customers"*), followed by POST methods which are usually used for creating resources (e.g., *"creating a new customer"*). The DELETE, PUT, and PATCH methods are also used for removing (e.g., *"delete a customer by id being «id»"*), replacing (e.g., *"replace a customer by id being «id»"*), and partially updating (e.g., *"update a customer by id being «id»"*) a resource.

Figure 6 also represents the distribution of number of segments in the operations[9] as well as the number of words in the generated canonical templates. As shown in Figure 6, many of the operations consist of less than 14 segments by 4 being the most common. Given the typical number of segments in the operations, Neural Machine Translation (NMT)-based approaches can be used for the generation of canonical sentences [29, 30]. On the other hand, the canonical sentences in the *API2CAN* dataset are longer. The reason behind having such lengthier utterances is the existence of parameters, and operations with more parameters tend to be lengthier. However, given the maximum length of canonical sentences, NMT-based approaches can still perform well [30].

4 NEURAL CANONICAL SENTENCE GENERATION

Neural Machine Translation (NMT) systems are usually based on encoder-decoder architecture to directly translate a sentence in one language to a sentence in a different language. As shown in Figure 7, generating a canonical template for a given operation can be also considered as a translation task. As such, the operation is encoded into a vector, and the vector is next decoded into an annotated canonical template. However, the main challenge in building such a translation model is the lack of a large training dataset. Since deep-learning models are data thirty, training requires a very large and diverse set of training samples (ideally millions of pairs of operations and their associating user utterances). As mentioned in the previous section, we automatically generated a dataset called *API2CAN*. However, such a dataset is still not large enough for training sequence-to-sequence models.

Having a large set of training samples requires a very large diverse set of operations as well. However, such a large set of APIs and operations is not available. One of the serious repercussions of the lack of such a set of operations is that training samples lack a very large number of possible words that can possibly appear in the operations (but did not appear in the training dataset). As a result, the models trained on such datasets will face many out-of-vocabulary words at runtime. To address this issue, we propose a delexicalization technique called *resource-based delexicalization*. As such, we reduce the impact of the out-of-vocabulary problem and force the model to learn the pattern of translating resources in an operation to a canonical template (instead of translating a sequence of words).

4.1 Resources in REST

In RESTful design, primary data representation is called *resource*. A resource is an object with a type, associated data, relationships to other resources, and a set of HTTP verbs (e.g., GET, POST) that operate on it. Designing RESTful APIs often involves following conventions in structuring URIs (endpoints) and naming resources. Examples include using plural nouns for naming resources, using the "GET" method for retrieving a resource and using the "POST" method for creating a new resource.

In RESTful design, resources can be of various types. Most commonly, a resource can be a document or a collection. A document, which is also called *singleton resource*, represents a single instance of the resource. For example, "`/customers/{customer_id}`" represents a customer that is identified by a path parameter ("`customer_id`"). On the other hand, a *collection resource* represents all instances of a resource type such as "`/customers`". Resources can be also nested. As such, a resource may also contain a sub-collection ("`/customers/{customer_id} /accounts`"), or a singleton resource (e.g., "`/customers/{customer_id} /accounts/{account_id}`"). In RESTful design, CRUD actions (create, retrieve, update and delete) over resources are shown by HTTP verbs (e.g., GET, POST). For example, "`GET /customers`" represents the action of getting the list of customers, and "`POST /customers`" indicates the action of creating a new customer. However, some actions might not fit into the world of conventional CRUD operations. In such cases, *controller* resources are used. *Controller* resources are like executable functions, with inputs and return-values. REST APIs rely on *action controllers* to perform application specific actions that cannot be logically mapped to one of the standard HTTP verbs. For example, an operation such as "`GET /customers/{customer_id}/activate`" can be used to activate a customer. Moreover, while it is unconventional, adjectives also are occasionally used for filtering resources. For example, "`GET /customers/activated`" means getting the list of all activated customers. In this paper, such adjectives are called *attribute controllers*.

[9] For example, "`GET /customers/{customer_id}`" has two segments: "`customers`" and "`{customer_id}`"

While above-mentioned principles are followed by many API developers, there are still many APIs violate these principles. By manually exploring APIs and prior works [5, 14, 15], we identified some unconventional resource types used in designing operations as summarized in Table 3. A common drift from RESTful principles is the use of programming conventions in naming resources (e.g., "createActor", "get_customers"). Aggregation functions (e.g., sum, count) and expected output format of an operation (e.g., "json", "tsb", "txt") are also used in designing endpoints. Words similar to "search" (e.g. "query", "item-search") are used to indicate that the operation looks for resources based on given criteria. Moreover, collections are occasionally filtered/sorted by using keywords such as "filtered-by", "sort-by", or appending a resource name to "By" (e.g., "ByName", "ByID"). Segments in the endpoints may also indicate API versions (e.g., v1.12), or authentication endpoints (e.g., auth, login). Even though the aforementioned types of resources are against the conventional design guidelines of RESTful design, they are important to detect since still they are used by API developers in practice.

Table 3: Resource Types

Resource Type	Example
Collection	`/customers`
Singleton	`/customers/{customer_id}`
Action Controller	`/customers/{customer_id}/activate`
Attribute Controller	`/customers/activated`
API Specs	`/api/swagger.yaml`
Versioning	`/api/v1.2/search`
Function	`/AddNewCustomer`
Filtering	`/customers/ByGroup/{group-name}`
Search	`/customers/search`
Aggregation	`/customers/count`
File Extension	`/customers/json`
Authentication	`/api/auth`

4.2 Resource-based Delexicalization

In resource-based delexicalization, the input (API call) and output (canonical template) of the sequence-to-sequence model are converted to a sequence of resource identifiers as shown in Figure 7. This is done by replacing mentions of resources (e.g., customers, customer) with a corresponding resource identifier (e.g., Collection_1). A resource identifier consists of two parts: (i) the type of resource and (ii) a number n which indicates n-th occurrence of a resource type in a given operation. This number later is used in the lexicalization of the output of the sequence-to-sequence model to generate a canonical template.

To detect resource types, we used the *Resource Tagger* shown in Algorithm 1. We convert the raw sequence of words in a given operation (e.g., "`GET /customers/{customer_id}/accounts`") to a sequence of resource identifiers (e.g., "`get Collection_1 Singleton_1 Collection_2`"). Likewise, mentions of resources in the canonical templates are replaced with their corresponding resource identifiers (e.g., "get all `Collection_1` for the `Collection_2` with `Singleton_1` being `Singleton_1`"). The intuition behind the conversions is to help the model to focus on translating a sequence of resources instead of words.

Algorithm 1: Resource Tagger

Input : *segments* of the operation
Result: List of resources

```
resources ⟵ [];
i ⟵ size(segments);
for i ← length(segments) down to 1 do
    current ⟵ segments[i];
    resource ⟵ new Resource();
    resource.name ⟵ current;
    previous ⟵ ϕ;
    if i > 1 then
        previous ⟵ segments[i − 1];
    end
    resource.type ⟵ "Unknown";
    if current is a path parameter then
        if previous is a plural noun and an identifier then
            resource.type ⟵ "Singleton";
            resource.collection ⟵ previous;
        else
            resource.type ⟵ "Unknown Param";
        end
    else
        if current starts with "by" then
            resource.type ⟵ "Filtering";
        else if current in ["count", "min", ...] then
            resource.type ⟵ "Aggregation";
        else if current in ["auth", "token", ...] then
            resource.type ⟵ "Authentication";
        else if current in ["pdf", "json", ...] then
            resource.type ⟵ "File Extension";
        else if current in ["version", "v1", ...] then
            resource.type ⟵ "Versioning";
        else if current in ["swagger.yaml", ...] then
            resource.type ⟵ "API Specs";
        else if any of ["search", "query", ...] in current then
            resource.type ⟵ "Search";
        else if current is a phrase and starts with a verb then
            resource.type ⟵ "Function";
        else if current is a plural noun then
            resource.type ⟵ "Collection"
        else if current is a verb then
            resource.type ⟵ "Action Controller";
        else if current is an adjective then
            resource.type ⟵ "Attribute Controller";
        end
        resources.append(resource);
end
return reversed(resources)
```

In the time of using the model for generating canonical templates, the tagged resource identifiers are replaced with their corresponding resource names (e.g., Collection_2 ⟶ customers). Meanwhile, in the process of replacing resource tags, occasionally grammatical errors might happen such as having plural nouns instead of singular forms. To make the final generated canonical template more robust, we used LanguageTool [10] (an

[10] https://languagetool.org

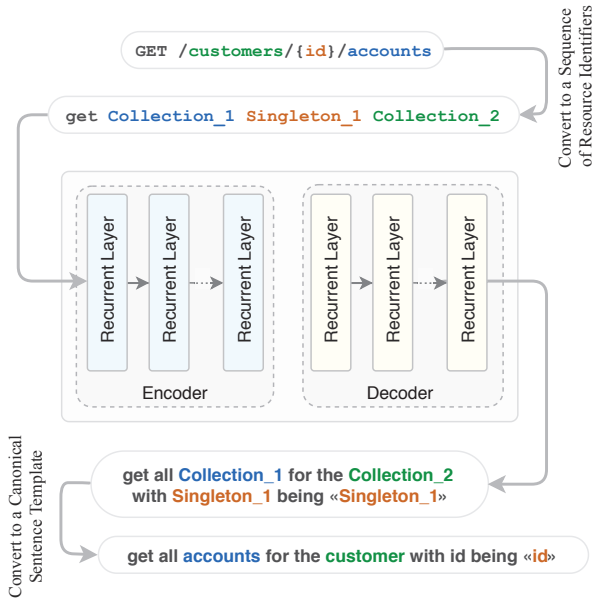


Figure 7: Canonical Template Generation via Resource-based Delexicalization

open-source tool for automatically detecting and correcting linguistically errors) to correct linguistic errors in the generated canonical templates.

5 PARAMETER VALUE SAMPLING

To obtain canonical utterances, values must be sampled for the parameters (placeholders) inside a given canonical template. The sampled values help to generate canonical utterances which are understandable sentences without any placeholders. Canonical utterances can be paraphrased later either automatically or manually by crowd-workers to diversify the training samples. In this section, we investigate how values can be sampled for parameters of REST APIs. More specifically, we identified five main sources as follows.

(1) **Common Parameters.** Parameters such as identifiers (e.g., customer_id), emails, and dates are ubiquitous in REST APIs. We built a set of such parameters paired with values. As such, a short random string or numeric value is generated for identifiers based on the parameter data type. Likewise, mock email addresses and dates are generated automatically.

(2) **API Invocation.** By invocation of API methods that return a list of resources (e.g., "`GET /customers`"), we can obtain a large number of values for various attributes (e.g., customer names, customer ids) of the resource. Such values are reliable since they correspond to real values of entities in the retrieved resources. Thus they can be used reliably to generate canonical utterances out of canonical templates.

(3) **OpenAPI Specification.** An OpenAPI specification may include an example or default values[11] for parameters of each operation. Since these values are generated by API owners, they are reliable. Moreover, API specification specifies the data-types of parameters. This can also be used to automatically generate values for parameters in the absence of example and default values. For example, in the case of *enumeration types* (e.g., gender ⟶ [MALE, FEMALE]), one of the elements is randomly selected as a parameter value. In the case of numeric parameters (e.g. size), a random number is generated within the specified range (e.g., between 1 to 10) in the API specification. Likewise, for the parameters whose values follow regular expressions (e.g., "[0-9]%" indicates a string that has a single-digit before a percent sign), random sequences are generated to fulfill the given pattern in the API specification (e.g., "8%").

(4) **Similar Parameters.** Having a large set of API specifications, example values can be found from similar parameters (sharing the same name and datatype). This can be possible by processing parameters of API repositories such as OpenAPI directory.

(5) **Named Entities.** Knowledge graphs provide information about various entities (e.g., cities, people, restaurants, books, authors). Examples of such knowledge graphs include for Freebase [31], DBpedia [32], Wikidata[33], and YAGO [34]. For a given entity type such as *"restaurant"* in the restaurant domain, these knowledge graphs might contain numerous entities (e.g., KFC, Domino's). Such knowledge bases can be used to sample values for a given parameter if the name of the parameter matches an entity type. In this paper, we use Wikidata to sample values for entity types. Wikidata is a knowledge graph which is populated by processing Wikimedia projects such as Wikipedia.

[11] An example illustrates what the value is supposed to be for a given parameter. But a default value is what the server uses if the client does not provide the value.

6 EXPERIMENTS & RESULTS

Before delving into the experiments, we briefly explain the training process in the case of using neural translation methods. We trained the neural models using the Adam optimizer [35] with an initial learning rate of 0.998, a dropout of 0.4 between recurrent layers (e.g., LSTM, BiLSTM), and a batch size of 512. It is worth noting that the hyper-parameters are initial configurations set based on the size of the dataset and values suggested in the literature, and finding optimized values requires further studies. Furthermore, in case of not using delexicalization, we also populate word embeddings of the model with GloVe [36].

In the time of translation, we used beam search with a beam size of 10 to obtain multiple translations for a given operation, and then the first translation with the same number of placeholders as the number of the parameters in the given operation is considered as its canonical template. Moreover, we replaced the generated *unknown* tokens with the source token that had the highest attention weight to avoid the out-of-vocabulary problem.

6.1 Translation Methods

We trained translation models using different sequence-to-sequence architectures and we also built a rule-based translator as described next. Given the size of the API2CAN dataset, we configured the models using two layers for both encoding and decoding parts at the most.

GRU. This model consists of two layers (each having 256 units) of Gated Recurrent Units (GRUs) [37] for both encoding and decoding layers using the attention mechanism [38].

Table 4: Excerpt of Transformation Rules

#		Resources Sequence	Transformation Rule
1	***Rule***	`GET /{c}/`	get list of {c.name}
	Example	`GET /customers`	get list of customers
2	***Rule***	`DELETE /{c}/`	delete all {c.name}
	Example	`DELETE /customers`	delete all customers
3	***Rule***	`GET /{c}/{s}/`	get the {*singular*(*c.name*)} with {s.name} being {s.name}
	Example	`GET /customers/{id}`	get the customer with id being <id>
4	***Rule***	`DELETE /{c}/{s}/`	delete the {*singular*(*c.name*)} with {s.name} being <{s.name}>
	Example	`DELETE /customers/{id}`	delete the customer with id being <id>
6	***Rule***	`PUT /{c}/{s}/`	replace the {*singular*(*c.name*)} with {s.name} being <{s.name}>
	Example	`PUT /customers/{id}`	replace the customer with id being <id>
7	***Rule***	`GET /{c}/{a}/`	get {a.name} {*singular*(*c.name*)}
	Example	`GET /customers/first`	get first customer
8	***Rule***	`GET /{c1}/{s}/{c2}/`	get the list of {c2.name} of the {*singular*(*c1.name*)} with {s.name} being {s.name}
	Example	`GET /customers/{id}/accounts`	get the list of accounts of the customer with id being <id>

LSTM. This model consists of two layers (each having 256 units) of two layers of LSTM for both encoding and decoding using the attention mechanism [38].

CNN. We also built a sequence-to-sequence model based on Convolutional Neural Network (CNN) as proposed in [39]. In particular, we used 3x3 convolutions (one layer of 256 units) with the attention mechanism [38].

BiLSTM-LSTM. This model consists of two layers (each having 256 units) of Bidirectional Long-Short Term Memory (BiLSTM) [40] for encoding, and two layers (each having 256 units) of Long-Short Term Memory (LSTM) [41] for the decoder using the attention mechanism [38].

Transformer. The Transformer architecture [42] has been shown to perform very strong in machine translation tasks [43, 44]. We used the Transformer model implemented by OpenNMT [45] using the same hyper-parameters as the original paper [42]. For an in-depth explanation of the model, we refer the interested reader to the original paper [42].

Rule-based (RB) Translator. Based on the concept of resource in REST APIs, we also built a rule-based translation system to translate operations to canonical templates (shown in Algorithm 2). First, the algorithm extracts the resources of a given operation based on the resource types extracted by the *Resource Tagger* algorithm (see Algorithm 1). Next, the algorithm iterates over an ordered set of *transformation rules* to transform the operation to a canonical template. A *transformation rule* is a handcrafted Python function which is able to translate a specific HTTP method (e.g., GET) and sequence of resource types (e.g., a *collection* resource followed by a *singleton* resource) to a canonical template. We created 33 *transformation rules* by the time of writing this paper, some of which are listed in Table 4. In this table, `{c}`, `{s}`, and `{a}` stands for collection, singleton, and attribute controller respectively. And the *singular*(.) function returns the singular form of a given name. For instance, in case of an operation like "`GET /customers`", given that the bot user requests a collection of customers, the provided transformer (rule number 1 in Table 4) is able to generate a canonical template as *"get the list of customers"*. Following Python function also presents the *transformation rule* implementation which is able to translate such operations (a single collection resource when the HTTP method is "GET"):

```
def transform(resources, verb):
 if verb != "GET" or len(resources) != 1:
    return
 if resources[0].type != "Collection":
    return
 collection = resources[0]
 return "get the list of {}".format(collection.name)
```

A transformer is written based on the assumption that a sequence of resource types has special meaning. For example, considering "`GET /customers/{id}/accounts`" and "`GET /users /{user_id}/aliases`", both operations share the same HTTP verbs and sequence of resource types (a singleton followed by a collection). In such cases, possible canonical templates are "*get accounts of a customer when its id is «id»*" and "*get aliases of a user when its user id is «user_id»*". Thus such a sequence of resource types can be converted to a rule like: *"get {collection} of a {singleton.collection} when its {singleton.name} is «{singleton.name}»"*; in which "{}" represents placeholders and *singleton.collection* represents the name of the collection for the given singleton resource (e.g., customers, users). Thus adding a new transformation rule would mean generalizing a specific sequence of resources types that is not considered in the existing translators. However, as discussed earlier, since many APIs do not conform to the RESTful principles, creating a comprehensive set of transformation rules is very challenging.

6.2 Canonical Utterance Generation

Quantitative Analysis. For each of the aforementioned NMT architectures, we trained models with and without using the

Algorithm 2: Rule-Based Translator

Input : *operation*, *transformation_rules* written by experts
Result: A canonical template

```
resources ⟵ ResourceTagger(operation);
foreach t ∈ transformation_rules do
    canonical ⟵ t.transform(resources, operation.verb);
    if canonical ≠ ϕ then
        param_clause ⟵ to_clause(operation.parameters) ;
        canonical ⟵ canonical + " " + param_clause ;
        return canonical;
    end
end
return ϕ
```

proposed *resource-based delexicalization* approach as described in Section 4.2. In these experiments, we did not tune any hyper parameters and trained the models on the training dataset. For each baseline, we saved the model after 10000 steps and used the model with the minimum perplexity based on the validation set to compare with other configurations. Table 5 presents the performance of each model in terms of machine translation metrics: bilingual evaluation understudy (BLEU) [46], Google's BLEU Score (GLEU) [47], and Character n-gram F-score (CHNF) [48].

In the case of using the RB-Translator, hand-crafted transformation rules are able to generate canonical templates for 26% of the operations. Creating such transformation rules is very challenging for lengthy operations as well as those not following RESTful principles. We did not include RB-Translators' performance in Table 5 because the results are not comparable to the rest. Our experiments indicate that RB-Translator performs reasonably well (BLEU=0.744, GLEU=0.746, and CHRF=0.850). However, the BiLSTM-LSTM model built on the proposed dataset using the resource-based delexicalization technique outperforms the RB-Translator (BLEU=0.876, GLEU=0.909, and CHRF=0.971), ignoring the operations which RB-Translator could not translate. As experiments indicate, *Delexicalized BiLSTM-LSTM* outperforms the rest of the translation systems, and resource-based delexicalization improves the performance of NMT systems by large.

Table 5: Translation Performance

Translation-Method	BLEU	GLEU	CHRF
Delexicalized BiLSTM-LSTM	**0.582**	**0.532**	**0.686**
Delexicalized Transformer	0.511	0.462	0.619
Delexicalized LSTM	0.503	0.470	0.652
Delexicalized CNN	0.507	0.458	0.601
Delexicalized GRU	0.481	0.450	0.623
BiLSTM-LSTM	0.318	0.266	0.421
Transformer	0.295	0.248	0.397
LSTM	0.278	0.226	0.381
CNN	0.271	0.220	0.379
GRU	0.251	0.198	0.347

Qualitative Analysis. Table 6 gives a few examples of canonical templates generated by the proposed translator (Delexicalized BiLSTM-LSTM). While the machine-translation metrics do not show very strong translation performance in Table 5, our manual inspections revealed that these metrics do not reflect the actual performance of the proposed translators. Therefore, we conducted another experiment to manually evaluated the translated operations. For this reason, we asked two experts to rate the generated canonical templates manually using a Likert scale (in a range of 1 to 5 with 5 showing the most appropriate canonical sentence). In the experiment, the experts were given pairs of generated canonical utterances and operations (including the description of the operation in the API specification). Next, they were asked to rate the generated canonical templates in a range of 1 to 5.

Figure 8 shows the Likert assessment for the best performing models in Table 5. Based on this experiment, canonical templates generated by RB-Translator are rated 4.47 out 5, and those of the delexicalized BiLSTM-LSTM are rated 4.06 out of 5 (by averaging the scores given by the annotators). The overall Kappa test showed a high agreement coefficient between the raters by Kappa being 0.86 [49]. Based on manual inspections, as also shown in Table 6, we observed that when APIs are designed based on the RESTful principles the delexicalized Delexicalized performs as good as RB-Translator.

Figure 8 also shows how the automatically generated dataset (*API2CAN*) represents their corresponding operations. Based on the rates given by the annotators, the dataset (training part) is also of decent quality while being noisy, indicating that the proposed set of heuristics for generating the dataset are well-defined. Given the promising quality of generated canonical templates, we concluded that the noises in the dataset can be ignored. However, yet it is desirable to manually clean the dataset.

Table 6: Examples of Generated Canonical Templates

	Sample
Operation	`GET /v2/taxonomies/`
Canonical	fetch all taxonomies
Operation	`PUT /api/v2/shop_accounts/{id}`
Canonical	update a shop account with id being <id>
Operation	`DELETE /api/v1/user/devices/{serial}`
Canonical	delete a device with serial being <serial>
Operation	`GET /user/ratings/query`
Canonical	get a list of ratings that match the query
Operation	`GET /v1/getLocations`
Canonical	get a list of locations
Operation	`POST /series/{id}/images/query`
Canonical	query the images of the series with id being <id>
Operation	`PUT /api /hotel /v0 /hotels /{hotelId} /rateplans/batch/$rates`
Canonical	set rates for rate plans of a hotel with hotel id being <hotelId>

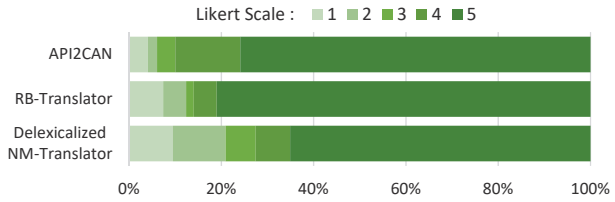


Figure 8: Assessment of Generated Canonical Templates

Error Analysis. Even though the proposed method outperforms the baselines, it still occasionally fails in generating high-quality canonical templates. Based on our investigations, there are three main sources of error in generating the canonical templates: (i) detecting resource types, (ii) translating APIs which do not conform to RESTful principles, and (iii) lengthy operations with many segments.

Detecting resource types requires natural language processing tools to detect parts of speech (POS) of a word (e.g., verb, noun, adjective), and to detect if a given noun is plural or singular (particularly for unknown words or phrases and uncountable nouns). However, these tools occasionally fail. Specifically, POS taggers are built for detecting parts of speech for words inside a sentence. Thus it is not surprising if they fail in detecting if a word like "rate" is a verb or noun in a given operation. For example, an operation like `GET /participation/rate` can indicate both "get the rate of participations" and "rate the participants". Another source of such issues is tokenization. It is common in APIs to concatenate words (e.g., whoami, addons, registrierkasseuuid, AddToIMDB). While it seems trivial for an individual to split these words, existing tools frequently fail. Such issues affect the process of detecting resources and consequently impact the generation of canonical templates negatively.

Unconventional API design (not conforming to RESTful principles) also extensively impacts the quality of generated canonical templates. Common drifts from RESTful principles includes using wrong HTTP verb (e.g., "`POST`" for retrieving information), using singular nouns for showing collections (e.g. `/customer`), adding non-resource parts to the path of the operation (e.g., adding *response format* like "json" in `/customers/json`. Since those API developers (who do not conform to design guidelines) follow their own thoughts instead of accepted rules, the automatic generation of canonical templates is challenging.

Lengthy operations (those with roughly more than 10 segments) naturally are rare in REST APIs. Such lengthy operations convey more complex intents than those with a lesser number of segments. As shown in Figure 6, unfortunately, such operations are also rare in the proposed dataset (*API2CAN*), impacting translation of lengthy operations.

6.3 Parameter Value Sampling

This section provides an analysis of parameters in the RESTful APIs and evaluates the proposed parameter sampling approach which is used for generating canonical utterances out of canonical templates. To this end, we processed API specifications which are indexed in OpenAPI Directory. Based on our analysis, the dataset contains 145971 parameters in total, which indicates that an operation has 8.5 parameters on average.

Figure 9 presents statistics of parameters in the whole list of API specifications in the OpenAPI Directory. As shown in the right-hand pie chart, most of the parameters are located in the

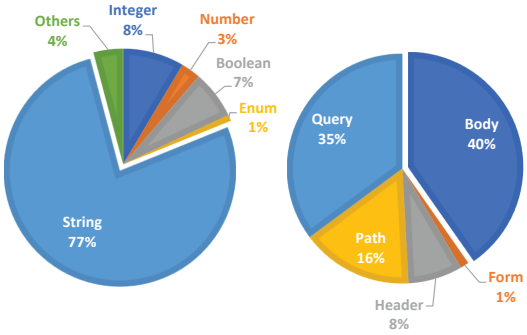


Figure 9: Parameter Type and Location Statistics

payload (body) of APIs, followed by query and path parameters. Figure 9 also shows the percentages of parameter data types in the collection with *strings* being the most common type of parameters. About 1.5% of string parameters are defined by regular expressions, and 4.8% of them can be associated with an entity type[12]. String parameters are followed by integers, booleans, numbers, and enumerations. Moreover, some parameters are left without any type, or they are given general parameter types such as *"object"* without any schemes. These parameters are combined together in the left-hand pie chart in Figure 9 with a single label– *"others"*. Moreover, 28% of parameters are required parameters (not optional), 10.6% of parameters have not assigned any value in the API specifications, and 26% of all parameters are identifiers (e.g., id, UUID). Thus, sampling values is required only for less than 10.6% of parameters (those without any values). In particular, value sampling for string parameters requires more attention. That is because string parameters are widely used, and they are more difficult to automatically be assigned values in comparison to other types of parameters (e.g., integers, enumerations).

To evaluate how well the proposed method generates sample values for parameters, we conducted an experiment. Since generating sample values for data types such as numbers and enumerations is straightforward, we only considered string parameters in this experiment. To this end, we randomly selected 200 parameters and asked an expert to annotate if a sampled value is appropriate for the given value or not. The results indicate that 68 percent of sampled values are appropriate for given parameters. The main reason for inappropriate sampled values is noises in the API specifications. For instance, developers occasionally describe the parameters in the *example* part instead of the *description* part of the documentation. For instance, for a string parameter like "`customer_id`", the example part may be filled by "a valid customer id". Moreover, sometimes the same parameter name is used in different contexts for different purposes. For example, the parameter name like *"name"* which can be used for representing the name of a person, school, or any object.

7 CONCLUSION & FUTURE WORK

This paper aimed at addressing an important shortcoming in current approaches for acquiring canonical utterances. In this paper, we demonstrated that the generation of canonical utterances can be considered as a machine translation task. As such, our work also aimed at addressing an important challenge in training supervised neural machine translators, namely the lack

[12]We looked up the parameter name in Wikidata to find if there is associating entity type

of training data for translating operations to canonical templates. By processing a large set of API specifications and based on a set of heuristics, we build a dataset called *API2CAN*. However, deep-learning-based approaches require larger sets of training samples to train domain-independent models. Thus, by formalizing and defining resources in REST APIs, we proposed a delexicalization technique to convert an operation to a tagged sequence of resources to help sequence-to-sequence models to learn from such a dataset. In addition, we showed how parameter values can be sampled to feed placeholders in a canonical template and generate canonical utterances. We also gave a systematic analysis of web APIs and their parameters, indicating the importance of string parameters in automating the generation of canonical utterances.

In our future work, we will be working on improving the dataset (*API2CAN*). Moreover, given that fulfilling complex intents usually requires a combination of operations [8, 50], we will be working on compositions between operations. To achieve this, it is required to detect the relations between operations and generate canonical templates for complex tasks (e.g., tasks requiring conditional operations or compositions of multiple operations). In future work, we will target these problems, together with many other exciting opportunities as extensions to this work.

ACKNOWLEDGEMENTS

This research was supported fully by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project DP1601104515).

REFERENCES

[1] Y. Su, A. H. Awadallah, M. Khabsa, P. Pantel, M. Gamon, and M. Encarnacion, "Building natural language interfaces to web apis," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17. New York, NY, USA: ACM, 2017, pp. 177–186. [Online]. Available: http://doi.acm.org/10.1145/3132847.3133009

[2] P. Babkin, M. F. M. Chowdhury, A. Gliozzo, M. Hirzel, and A. Shinnar, "Bootstrapping chatbots for novel domains," in *Workshop at NIPS on Learning with Limited Labeled Data (LLD)*, 2017.

[3] M. Vaziri, L. Mandel, A. Shinnar, J. Siméon, and M. Hirzel, "Generating chat bots from web api specifications," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2017. New York, NY, USA: ACM, 2017, pp. 44–57. [Online]. Available: http://doi.acm.org/10.1145/3133850.3133864

[4] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, and G. Tremblay, "Are restful apis well-designed? detection of their linguistic (anti)patterns," in *Service-Oriented Computing*, A. Barros, D. Grigori, N. C. Narendra, and H. K. Dam, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 171–187.

[5] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "Rest apis: A large-scale analysis of compliance with principles and best practices," in *Web Engineering*, A. Bozzon, P. Cudre-Maroux, and C. Pautasso, Eds. Cham: Springer International Publishing, 2016, pp. 21–39.

[6] F. Palma, J. Gonzalez-Huerta, M. Founi, N. Moha, G. Tremblay, and Y.-G. GuÃĪhÃĪneuc, "Semantic analysis of restful apis for the detection of linguistic patterns and antipatterns," *International Journal of Cooperative Information Systems*, p. 1742001, 05 2017.

[7] M.-A. Yaghoub-Zadeh-Fard, B. Benatallah, M. Chai Barukh, and S. Zamanirad, "A study of incorrect paraphrases in crowdsourced user utterances," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 295–306. [Online]. Available: https://www.aclweb.org/anthology/N19-1026

[8] G. Campagna, R. Ramesh, S. Xu, M. Fischer, and M. S. Lam, "Almond: The architecture of an open, crowdsourced, privacy-preserving, programmable virtual assistant," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017, pp. 341–350. [Online]. Available: https://doi.org/10.1145/3038912.3052562

[9] T.-H. K. Huang, W. S. Lasecki, and J. P. Bigham, "Guardian: A crowd-powered spoken dialog system for web apis," in *Third AAAI conference on human computation and crowdsourcing*, 2015.

[10] M. Negri, Y. Mehdad, A. Marchetti, D. Giampiccolo, and L. Bentivogli, "Chinese whispers: Cooperative paraphrase acquisition," in *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*. Istanbul, Turkey: European Language Resources Association (ELRA), May 2012, pp. 2659–2665. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2012/pdf/772_Paper.pdf

[11] Y. Wang, J. Berant, and P. Liang, "Building a semantic parser overnight," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, Jul. 2015, pp. 1332–1342. [Online]. Available: https://www.aclweb.org/anthology/P15-1129

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," 1999.

[13] C. Pautasso, "Restful web services: principles, patterns, emerging technologies," in *Web Services Foundations*. Springer, 2014, pp. 31–51.

[14] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of rest patterns and antipatterns: A heuristics-based approach," in *Service-Oriented Computing*, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 230–244.

[15] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Are rest apis for cloud computing well-designed? an exploratory study," in *Service-Oriented Computing*, Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, Eds. Cham: Springer International Publishing, 2016, pp. 157–170.

[16] J. Weizenbaum, "Eliza—a computer program for the study of natural language communication between man and machine," vol. 9, no. 1. New York, NY, USA: ACM, Jan. 1966, pp. 36–45. [Online]. Available: http://doi.acm.org/10.1145/365153.365168

[17] W. Y. Wang, D. Bohus, E. Kamar, and E. Horvitz, "Crowdsourcing the acquisition of natural language corpora: Methods and observations," in *2012 IEEE Spoken Language Technology Workshop (SLT)*, Dec 2012, pp. 73–78.

[18] A. Ravichander, T. Manzini, M. Grabmair, G. Neubig, J. Francis, and E. Nyberg, "How would you say it? eliciting lexically diverse dialogue for supervised semantic parsing," in *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, 2017, pp. 374–383.

[19] J. Fang and J. Kessler, "System and method to acquire paraphrases," Jun. 6 2017, uS Patent 9,672,204.

[20] K. Dhamdhere, K. S. McCurley, R. Nahmias, M. Sundararajan, and Q. Yan, "Analyza: Exploring data with conversation," in *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, ser. IUI '17. New York, NY, USA: ACM, 2017, pp. 493–504. [Online]. Available: http://doi.acm.org/10.1145/3025171.3025227

[21] N. Duan, D. Tang, P. Chen, and M. Zhou, "Question generation for question answering," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 866–874. [Online]. Available: https://www.aclweb.org/anthology/D17-1090

[22] L. Dong, J. Mallinson, S. Reddy, and M. Lapata, "Learning to paraphrase for question answering," *arXiv preprint arXiv:1708.06022*, 2017.

[23] J. Mallinson, R. Sennrich, and M. Lapata, "Paraphrasing revisited with neural machine translation," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, vol. 1, 2017, pp. 881–893.

[24] R. Bapat, P. Kucherbaev, and A. Bozzon, "Effective crowdsourced generation of training data for chatbots natural language understanding," in *Web Engineering*, T. Mikkonen, R. Klamma, and J. Hernández, Eds. Cham: Springer International Publishing, 2018, pp. 114–128.

[25] S. A. Hasan, B. Liu, J. Liu, A. Qadir, K. Lee, V. Datla, A. Prakash, and O. Farri, "Neural clinical paraphrase generation with attention," in *Proceedings of the Clinical Natural Language Processing Workshop (ClinicalNLP)*, 2016, pp. 42–53.

[26] A. Gupta, A. Agarwal, P. Singh, and P. Rai, "A deep generative framework for paraphrase generation," *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[27] Q. Xu, J. Zhang, L. Qu, L. Xie, and R. Nock, "D-page: Diverse paraphrase generation," *CoRR*, vol. abs/1808.04364, 2018.

[28] F. Brad and T. Rebedea, "Neural paraphrase generation using transfer learning," in *Proceedings of the 10th International Conference on Natural Language Generation*, 2017, pp. 257–261.

[29] J. Pouget-Abadie, D. Bahdanau, B. van Merriënboer, K. Cho, and Y. Bengio, "Overcoming the curse of sentence length for neural machine translation using automatic segmentation," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 78–85. [Online]. Available: https://www.aclweb.org/anthology/W14-4009

[30] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 103–111. [Online]. Available: https://www.aclweb.org/anthology/W14-4012

[31] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: A collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 1247–1250.

[Online]. Available: http://doi.acm.org/10.1145/1376616.1376746
[32] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer *et al.*, "Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.
[33] D. Vrandečić and M. Krötzsch, "Wikidata: A free collaborative knowledgebase," *Commun. ACM*, vol. 57, no. 10, pp. 78–85, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629489
[34] T. Rebele, F. Suchanek, J. Hoffart, J. Biega, E. Kuzey, and G. Weikum, "Yago: A multilingual knowledge base from wikipedia, wordnet, and geonames," in *The Semantic Web – ISWC 2016*, P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil, Eds. Cham: Springer International Publishing, 2016, pp. 177–185.
[35] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.
[36] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://www.aclweb.org/anthology/D14-1162
[37] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, 2014, pp. 1724–1734. [Online]. Available: https://www.aclweb.org/anthology/D14-1179/
[38] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 1412–1421. [Online]. Available: https://www.aclweb.org/anthology/D15-1166
[39] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, pp. 1243–1252. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305381.3305510
[40] A. Graves, S. Fernández, and J. Schmidhuber, "Bidirectional lstm networks for improved phoneme classification and recognition," in *Artificial Neural Networks: Formal Models and Their Applications – ICANN 2005*, W. Duch, J. Kacprzyk, E. Oja, and S. Zadrożny, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 799–804.
[41] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. USA: Curran Associates Inc., 2017, pp. 6000–6010. [Online]. Available: http://dl.acm.org/citation.cfm?id=3295222.3295349
[43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019.
[44] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," *URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/languageunsupervised/language understanding paper. pdf*, 2018.
[45] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proceedings of ACL 2017, System Demonstrations*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 67–72. [Online]. Available: https://www.aclweb.org/anthology/P17-4012
[46] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, pp. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135
[47] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.
[48] M. Popović, "chrF: character n-gram f-score for automatic MT evaluation," in *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 392–395. [Online]. Available: https://www.aclweb.org/anthology/W15-3049
[49] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica: Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.
[50] E. Fast, B. Chen, J. Mendelsohn, J. Bassen, and M. S. Bernstein, "Iris: A conversational agent for complex tasks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 473:1–473:12. [Online]. Available: http://doi.acm.org/10.1145/3173574.3174047