Improving Examples in Web API Specifications using Iterated-Calls In-Context Learning

Kush Jain
Carnegie Mellon University
United States
kdjain@andrew.cmu.edu

Kiran Kate
IBM Research
United States
kakate@us.ibm.com

Jason Tsay
IBM Research
United States
Jason.Tsay@ibm.com

Claire Le Goues
Carnegie Mellon University
United States
clegoues@cs.cmu.edu

Martin Hirzel
IBM Research
United States
hirzel@us.ibm.com

***Abstract*—Examples in web API specifications can be essential for API testing, API understanding, and even building chat-bots for APIs. Unfortunately, most API specifications lack human-written examples. This paper introduces a novel technique for generating examples for web API specifications. We start from in-context learning (ICL): given an API parameter, use a prompt context containing a few examples from other similar API parameters to call a model to generate new examples. However, while ICL tends to generate correct examples, those lack diversity, which is also important for most downstream tasks. Therefore, we extend the technique to iterated-calls ICL (ICICL): use a few different prompt contexts, each containing a few examples, to iteratively call the model with each context. Our intrinsic evaluation demonstrates that ICICL improves both correctness and diversity of generated examples. More importantly, our extrinsic evaluation demonstrates that those generated examples significantly improve the performance of downstream tasks of testing, understanding, and chat-bots for APIs.**

I. Introduction

Web Application Programming Interfaces (APIs) enable systems to communicate across a network [1], [2]. REpresentational State Transfer (REST) APIs have become the de facto standard for modern web applications [3]. This style enables clients and services to exchange information over HTTP. Large companies like Google, Amazon, and Apple expose services through REST APIs, including large enterprise services like Google Drive and Apple Authentication, as well as simpler services like REST Countries,[1] for querying information about a country.

REST APIs are commonly described using OpenAPI specifications [4]: one survey of communication service providers found that 73% of companies and 75% of suppliers use OpenAPI to describe their APIs.[2] OpenAPI specifications formalize the contract between API developer and API user, describing the structure of API requests and responses. Tools such as Redoc[3] and SwaggerUI[4] can automatically convert OpenAPI specifications into human-readable webpages, allowing developers to better understand these APIs. Additionally, specifications are commonly used in input validation [5], [6] and testing [7], [8].

Common downstream clients of OpenAPI specifications leverage realistic examples of OpenAPI parameters (when they exist) as a part of their workflow. Fuzzers [9], [10] use examples to guide API testing, producing fewer invalid requests and covering deeper code paths. Chat-bots [11], [12], [13] first build an underlying model of a system and then derive API calls from the natural language utterance. Recently-developed large language models (LLMs), like ChatGPT [14] and GPT-4 [15], benefit from using API parameter examples, and other LLMs use them for fine-tuning, as evaluated on dialog benchmarks [16]. API parameter examples can also improve human understanding, especially for novice users [17], [18].

However, despite their widespread adoption, most OpenAPI specifications lack API parameter examples (only 1,953 out of a dataset of 13,346 mined OpenAPI parameters have any examples). There has been some research on generating examples for OpenAPI specifications. Prior work follows two approaches: (i) extracting examples from API descriptions [10] or (ii) mining examples from knowledge bases [9], [19]. The goal of both approaches is to generate diverse and correct examples. Example correctness is important, as these examples serve as input to software testing and dialog systems. Conversely, example diversity is also important, as examples that differ from one another help testing increase its coverage and help chat-bots generalize their natural-language understanding. Both approaches to example generation are limited: mining examples only works for examples present in knowledge bases, while extracting examples from descriptions only works when the description explicitly enumerates parameter examples.

[1] https://restcountries.com

[2] https://inform.tmforum.org/features-and-opinion/the-status-of-open-api-adoption/

[3] https://github.com/Redocly/redoc

[4] https://github.com/swagger-api/swagger-ui

We present ICICL, which combines retrieval-based prompting [20] with iterated calls to in-context learning (ICL) to generate diverse and correct API parameter examples. ICICL leverages the ability of LLMs to generate realistic examples based on their pretraining. Unlike knowledge bases, LLMs are pretrained on large swaths of the internet, and thus have a strong prior of the world around them. We take as input the OpenAPI specification without examples and generate examples for all API parameters, regardless of whether examples exist on the internet or the descriptions specify example values. For correctness, we use greedy decoding (taking the highest probability token at each step) to generate one (likely) correct example. We perform postprocessing to only keep examples that are similar to our (likely) correct example. For diversity, we both increase temperature, and, unlike vanilla ICL, use iterated calls with multiple prompt contexts. One can increase temperature (smoothing the distribution of next token probabilities) to generate different model outputs. Additionally, we observe that the problem of example diversity is similar to the challenge of generating different model outputs, which is solved by ensembles [21] of different models. This observation leads us to use multiple prompt contexts, where each context consists of a different set of few-shot examples.

We evaluate ICICL, finding that it generates diverse, correctly typed examples. We further manually annotate a sample of 385 parameters and show that 75% of the generated examples are correct. We then demonstrate the usefulness of the generated examples in three downstream settings: fuzzing, dialogue benchmarks, and human API understanding, which we assess via an exploratory developer pilot. Our examples significantly improve performance in these tasks, improving branch coverage by 116%, dialog intent recognition by 3%, and dialog slot filling by 5%, compared to the original specifications.

To summarize, our core contributions are as follows:

- We identify adding examples as a *single* improvement to API specifications that benefits *several* downstream use cases (understanding, fuzzing, chat-bots).
- Inspired by how ensembles use multiple models to improve results, we introduce ICICL, a new technique for using LLMs to generate API examples. We combine retrieval-based prompting, multiple prompt contexts, and post-processing to produce diverse yet correct examples.
- We include an extensive experimental evaluation that quantifies the value of the generated examples for several use-cases. These include fuzz testing, chat-bots, and an exploratory study of developers' API understanding.

Our prompting, intrinsic, fuzzing, and exploratory study evaluation and code are at https://figshare.com/s/8eec881ddf8e6573f43f, including detailed reproduction instructions. We elide calls to internal company services in the prompting code, but release all other code. We are unfortunately unable to release our API parameter bank, intrinsic evaluation dataset, and SeqATIS dataset, as they are internal to the large technology company at which this work was conducted, but hope that the other elements of the artifact are informative for subsequent research.

Listing 1: Illustrative OpenAPI parameter from the Rest Countries API. Prior approaches struggle to generate correct examples for this API parameter; knowledge bases contain many false positives, and the description contains no examples.

```yaml
name: currency
description: Search by ISO 4217 currency code
in: path
required: true
schema:
    type: string
```

II. Motivating Example and Overview

Listing 1 shows an illustrative parameter for the `/currency` endpoint[5] of the REST Countries API. As with most OpenAPI parameters, this specification contains its name, a short description, and a type. However, it does not contain any example values, nor can example values easily be extracted from the description or name. To try the `/currency` endpoint, a developer would either need domain knowledge of ISO 4217 currency codes or would need to search for an example. Fuzzers also fail to cover deeper code paths for this endpoint, as they would start from a random sequence of bits and would only arrive at a valid ISO 4217 currency code by chance.

Two common approaches to generating example values, namely mining them from a knowledge base such as DBPedia or extracting examples from the description, would also fail here. While ISO 4217 is an entity in DBPedia (the knowledge base used by the state-of-the-art example generation tool, ARTE [9]), there are numerous other currency codes that are not ISO 4217, meaning that generated examples are semantically incorrect. ARTE [9] circumvents this by calling the API with examples to see if they are valid; however, this limits applicability to cases like fuzzing, which can send a large volume of requests to the API. The description also does not enumerate examples of currency codes that could be extracted.

Figure 1 gives an overview of ICICL. It first retrieves parameters from the API parameter bank that are similar to the parameter from the original API specification (step ①). Then it creates a prompt context by greedily selecting the top-most similar retrieved parameters for in-context learning (step ②). Following this, it uses the LLM with greedy decoding to obtain the greedy example, which has the highest confidence (step ③). It then creates multiple diverse prompt contexts, each of which includes the greedy example plus some retrieved parameters for in-context learning, selected to be similar but with some randomization (step ④), and uses iterated calls to the LLM with a higher temperature to obtain multiple diverse examples, one from each of the diverse prompt contexts (step ⑤). Then it creates a list

[5] https://restcountries.com/v2/currency

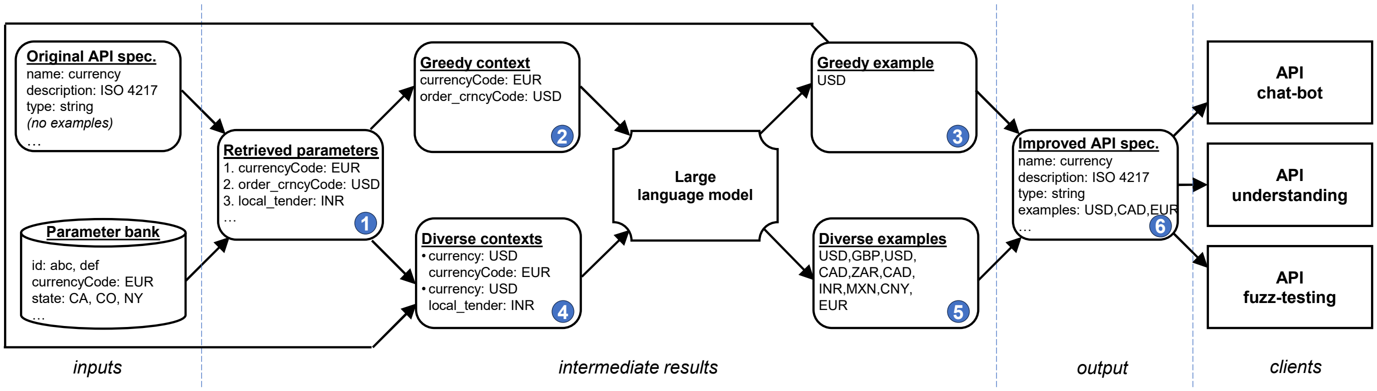

Fig. 1: Overview and running example of our approach. Circled numbers correspond to different steps in our approach.

of filtered examples that include the greedy example and some of the diverse examples, which it adds to the API specification (step 6).

Our approach performs well on the snippet in Listing 1, generating `USD`, `CAD`, and `EUR`, all valid currency examples.

III. IcIcl

IcIcl takes an API specification without parameter examples as input and returns an improved specification with those examples as output. Figure 1 outlines the approach. Offline, we create a parameter bank by mining parameters and examples, and pick an off-the-shelf LLM (Section III-A). Online, given a parameter in an OpenAPI specification, we retrieve relevant parameters from the parameter bank (Section III-B), build prompt contexts (Section III-C), and finally postprocess model output (Section III-D).

A. Offline: Mining Examples, Model Selection

We mine 1,236 OpenAPI specifications from API Guru [22] and Wittern et. al. [23]. This collection has OpenAPI specifications for popular enterprise applications such as Box, Google Drive, YouTube, and others. We parse the mined specifications to extract each API parameter and corresponding examples. Of 13,346 parameters, 1,953 have examples. We use these mined examples as our parameter bank (shown among the inputs on the left of Figure 1).

We use Falcon,[6] a 40B parameter model trained on one trillion tokens from the internet, for the LLM (middle of Figure 1). Falcon outperforms LLAMA, GPT-3, and MPT on the OpenLLM leaderboard. Falcon has also been extensively pretrained on code, which we hypothesize will help with type correctness. Using a large but not huge open-source model such as Falcon is representative of commercial settings that must balance cost and data exposure regulatory concerns. We model the task of example-generation as an instance of few-shot prompting, varying the prompt context to generate different examples.

B. Retrieving Relevant Parameters

Given an API parameter, we first seek a set of relevant similar parameters from the parameter bank (Figure 1: 1). We first extract the initial 50 characters from the API parameter description (our dataset has a median description length of 54 characters, with the first 50 characters concisely representing a parameter's purpose or function). At times, the full description is excessively verbose, with all other parameter information outside the description having a median length of 63 characters, thus truncating at 50 characters ensures that we do not overwhelm other important information. We append the exact name of the API parameter to the description, ensuring that the name of the API parameter is factored into any similarity computation. Lastly, we append the operation ID, which offers additional context about the operation associated with the parameter. For example, the parameter '`name`' has different meanings if the operation ID is '`getCountries`' or '`getUserByUsername`'. We use the concatenated string of the parameter description, parameter name, and operation ID as the *query* for retrieval.

We use BM25 [24] as the retrieval method, due to its high speed and accuracy [25]. BM25 calculates a weight of terms based on their frequency in both the query and the target documents. It then considers the term's prevalence across the entire parameter bank (intuitively infrequent terms discriminate better). When BM25 processes a parameter (such as 'currency' in the running example), it returns a similarity score for each API parameter in the parameter bank. These scores measure how closely each parameter in the parameter bank matches the parameter we are generating examples for. We leverage this distribution of similarity scores to craft prompt *contexts* (sets of few-shot examples for in-context learning).

C. Prompt Context Generation

Prompt context generation consists of two phases: eliciting the greedy example, and then constructing ten prompt contexts (of five shots each) to elicit diverse examples. We use a two-phase approach to improve both correctness and diversity of the generated examples.

[6] https://huggingface.co/tiiuae/falcon-40b

Listing 2: LLM prompt for currency code. We provide the parameter that is missing examples and five few-shot examples.

```
# Given an OpenAPI parameter, generate a unique
    example of the parameter.
input_0 = {
    "param_name": "currencyCode",
    "type": "string",
    "operation_id": "contractInfo",
    "description": "The currency code (ISO 4217)",
    "api_name": "beezup"
}
# must generate a unique currencyCode string
example_0 = "EUR"
...
input_6 = {
    "param_name": "currency",
    "type": "string",
    "operation_id": "v2Currency",
    "description": "Search by ISO 4217 currency code
        ",
    "api_name": "rest-countries"
}
# must generate a unique currency string
example_6 =
```

The first phase prompts the LLM with the top five retrieved parameters with the highest similarity to the query (Figure 1: ②) as returned by retrieval (Section III-B). Greedy decoding in an LLM simply picks the most likely token at each generation step, thus deterministically yielding the sequence of most-probably tokens. By leveraging greedy decoding, this step aims to produce a (likely) correct example ③. Our model yields `USD` as the greedy example (a correct currency code). By providing the greedy example in all prompt contexts, we ensure that the LLM, even at a higher temperature setting, generates examples that align with the original example.

The second phase improves example diversity by sampling from the distribution of similarity scores to generate 10 prompt contexts of five examples each ④. We take inspiration from ensembles [21], where multiple models produce different outputs that improve both the correctness and diversity of the resulting system. Our prompt contexts, each of which consists of a different set of API parameter examples (i.e., "shots"), are similar to the diverse models used in ensembles. We iteratively call the LLM with each prompt context with a higher temperature of 0.5 to generate 10 example candidates ⑤. The order of these calls does not matter; they can be parallelized or batched. In the running example, the calls return `USD`, `GPP`, `USD`, `CAD`, `ZAR`, `CAD`, `INR`, `MXN`, `CNY`, and `EUR`.

D. *Postprocessing*

We perform postprocessing to narrow these 10 example candidates down to 3 examples to add to the improved API specification (Figure 1: ⑥). First, we filter out all examples that do not match in type to the API parameter we are generating examples for. This is the earliest opportunity for this filter, and we do it right away given the importance of type compatibility. We then add the greedy example to the 10 example candidates and perform deduplication. We always include the greedy example in our set of three generated examples, as it is likely to be correct. Following this, we add all examples that the model generates multiple times to our set of three, and return this set if it contains at least three examples. For example, if the model generates the currency `CAD` twice, then we add it to the final set of three examples. If, at this point, there are fewer than three examples, we use BERT [26] to encode each example and the greedy example. We then select the most similar examples until we have three examples (illustrated by adding `EUR` in Figure 1).

This ensures that the generated examples are similar in format and content to the greedy example, improving their likelihood of being correct. We choose to favor correctness over diversity here, given its importance to downstream tasks (testing and chat-bots). Using BERT embeddings ensures that we are comparing the semantic similarity of each example to the greedy example, rather than doing a simple text-based match (which, in the case of currency codes, is less meaningful).

IV. Intrinsic Evaluation

While ultimately extrinsic evaluations (Section V) matter most for downstream clients, they are laborious to measure, so we used intrinsic evaluations for nimble iterative modeling. We evaluate examples generated by ICICL on intrinsic correctness and diversity. Specifically, we measure whether examples generated by ICICL are type correct, unique, and semantically diverse. We also hand-evaluated a smaller subset of examples for semantic correctness. We compare different components of ICICL across these metrics.

A. *Experimental Setup*

1) Dataset: We evaluate modeling approaches on a randomly sampled dataset of 1,000 OpenAPI parameters mined from mainstream services including but not limited to Box, Google Drive, and Gmail. We remove all parameters in the parameter bank from our set of API parameters prior to sampling. We also remove all Boolean and enum parameters (approximately 1,000 from the initial mined set) from our evaluation set, as predicting the values of these parameters is trivial. Due to computational cost, we do not run our intrinsic evaluation on the full final set of 13,346 parameters, instead focusing on a likely-representative random sample of 1,000 examples (approximately 1/13) of the dataset. This sampling is in line with prior work [27], [28], which sample a similar proportion of the dataset for evaluation. These include 668 string, 129 array, 106 integer, 34 number, 14 object, and 5 datetime types. The remaining 44 parameters come from a variety of other types including color, tuples, and None types.

2) Approach: We evaluate the efficacy of each component of our approach (adding retrieval, sampling from the distribution of similarity scores, and applying our postprocessing). This is equivalent to an ablation study: the final setting is the full approach, earlier settings remove components. We prompt the model as described in each settings and evaluate the generated examples using the metrics described below.

Static: Static refers to a static prompt of five parameter and example pairings for in-context learning. We also include the greedy example as part of the prompt and use a temperature of 0.5. Temperature corresponds to the level of randomness in text generation - temperature of 0 refers to sampling the most likely tokens, while higher temperature refers to sampling more diversely. We prompt the LLM 10 times to generate 10 examples and perform deduplication. Finally, we randomly select three examples to return to the user.

Retrieval: Retrieval refers to the greedy retrieval approach. Rather than sampling 10 prompt contexts from the distribution of similarity scores, we only use a single prompt context containing the five most similar parameters for prompting. In other words, this setting performs in-context learning with retrieval, but no iterated calls.

Retrieval (w/contexts): Our retrieval with context approach adds iterated calls with context sampling. Rather than selecting the five most similar examples for all 10 prompts, we build prompt contexts by randomly sampling from the distribution of similarity scores (similar parameters are more likely to be chosen than different parameters).

Retrieval (w/postprocessing): This is our final approach used in extrinsic evaluations (fuzzing, dialog, and exploratory usability study). We apply our postprocessing that filters out type-incorrect examples and selects examples that are similar to the greedy example. This helps ensure that our examples are correct, both in type and in semantic meaning (close to a generated example likely to be correct).

3) Metrics: We define the following set of metrics to benchmark various prompting approaches. The main factors we consider are example correctness and example diversity.

Type Correctness: Type correctness adheres to the strict definition of all generations from the LLM being the same type as the parameter. We use this strict definition to ensure all generations conform to the same example type. Recall that our intrinsic evaluation focuses on open-ended types (strings, numbers, arrays, objects, etc.) but not Boolean or enums (as generating values for types with small closed sets is trivial).

Uniqueness: Uniqueness refers to the ability of the LLM to generate three case-insensitive unique examples from 10 generations. Higher uniqueness values indicate more diverse LLM generated examples. For example, if all 10 generations are the same example, the uniqueness would be 0, otherwise if there are three unique examples it would be 1.

Diversity: Diversity is 1 minus mean cosine similarity between the BERT [26] embeddings of examples. We choose to use BERT embeddings over TF-IDF or BM25 embeddings, as BERT embeddings detect semantic similarity, while other approaches only detect overlap of tokens (syntactic similarity).

Example Correctness: Example correctness refers to generated examples matching the specification. We define correctness as examples that both satisfy preconditions specified in the natural language description of the parameter and have consistent format between all generated examples. Correct examples can be used in an API call to the API under test without 4xx or input validation errors. Unlike the other metrics, which are fully automated, this metric requires human effort. We manually annotate a randomly sampled subset of 385 out of our 1,000 sampled examples across all four settings, for 95% confidence in the correctness results.

B. Intrinsic Evaluation Results

TABLE I: Intrinsic evaluation metrics on 1,000 (columns Type, Unique, Both (type correct and unique), Div) and 385 (column Correct) randomly sampled examples. Each approach component improves type correctness, the proportion of unique examples, and overall correctness.

Setting	**Type**	**Unique**	**Both**	**Div**	**Correct**
static	97%	48%	47%	0.22	70.4%
retrieval	98%	55%	55%	0.20	73.2%
w/contexts	98%	66%	65%	**0.23**	65.7%
w/postprocessing	**99%**	**67%**	**67%**	0.19	**74.3%**

Table I shows the results from running various components of IcIcl on a selected OpenAPI parameters. We show how each component improves on the baseline.

We find that type correctness of generated examples is relatively strong across all approaches (varying from 97% to 99%). We hypothesize this is due to LLMs' extensive training on code, where type is important in generating the next token. However, we do notice that type correctness does increase as we add retrieval-based prompting and our postprocessing, which improves type correctness to 99%.

In terms of generating unique examples, we find that each step in our process improves upon the previous step. Retrieval and adding contexts see approximately a 10% improvement over the previous steps. Postprocessing improves uniqueness slightly, with 67% of examples generated having 3 examples. Cosine similarity between examples remains relatively stable across modes, with retrieval templating (third row) slightly improving example diversity. We do want examples to have consistent format, while still being diverse, likely resulting in lower diversity scores. The average Levenshtein edit distance on our dataset is 15 characters, suggesting the examples are still syntactically different from one another on average.

Example correctness remains relatively stable across all four settings (varying from 66% to 74%). The correctness of our final approach is higher than any intermediate approach. Note that our evaluation of example correctness is conservative: in order for an example to be correct, all generations need to satisfy preconditions and have consistent format. Even examples not labeled as correct can still be useful for developers (such as an example of a time parameter that is missing the timezone), meaning that the 74% correctness rate is likely an underestimate of the true utility of the examples. Overall, our approach is often correct, showing the promise that LLMs pose for usefully enhancing API specifications. The extrinsic evaluation in the following section shows that, despite not always being correct, synthetic examples benefit all three downstream clients we tried.

Listing 3: Fuzzing enhanced OpenAPI specification. Examples generated by IcIcl are both diverse and correct.

```
name: currency
description: Search by ISO 4217 currency code
required: true
schema:
    type: string
    enum:
    -   USD
    -   CAD
    -   EUR
example: USD
...
name: currency
description: Search by ISO 4217 currency code
required: true
schema:
    type: string
```

V. Extrinsic Evaluation

This section evaluates IcIcl on downstream tasks (clients on the right-hand side of Figure 1), namely software testing (Section V-A), API chatbots (Section V-B), and, by means of an exploratory pilot, human API understanding (Section V-C).

A. Software Testing

The goal of REST API testing is to find inputs that increase code coverage (and, ultimately, find bugs). At a high level, API fuzzers encode the schemas present in OpenAPI specifications, and use them to generate values for API endpoints. Coverage serves as a feedback mechanism: calls that increase coverage are saved for further mutation, while calls that do not are thrown out.

1) Dataset and fuzzers: We evaluate IcIcl for fuzzing using a dataset from Kim et. al. [10] consisting of both small and large APIs. We exclude the OMDB and Spotify APIs from that dataset, due to changes in both that make usage more challenging, and internal restrictions that block certain endpoints. This leaves seven widely-used REST API services — FDIC, REST Countries, ohsome, GenomeNexus, OCVN, LanguageTool, and YouTube. This previous dataset included 4 that were run as local instances — GenomeNexus, OCVN, LanguageTool, and YouTube — for the purposes of computing coverage. We therefore follow the previous evaluation and compute black-box performance on all 7 and coverage on the 4.

We evaluate using four popular fuzzers: EvoMaster [8], MoREST [7], RESTest [29], and RestTestGen [30]. We report results for each fuzzer along with the aggregated results across them all. We used a version of RESTest that includes ARTE [9], a state-of-the-art example generation approach, as part of its implementation. Hence, we refer to it as RESTest/ARTE below, helping show how IcIcl compares to and can complement ARTE.

2) Approach: To measure performance in the fuzzing context, we run IcIcl with each OpenAPI parameter that we extract from the fuzzing OpenAPI specifications. For each API parameter, we overload the specification with two options: the parameter with examples, and the parameter without examples, following Kim et al. [10]. Since most fuzzers do not directly use API examples, we had to use a work-around, where we encode the examples both using the example attribute and as an enum. Listing 3 shows how we encode these values (the generated examples are `USD`, `CAD` and `EUR`). We also include the original parameter, to allow for fuzzers to explore values outside these examples. This ensures the fuzzer can explore the example values and mutate existing example values by hitting the overloaded endpoint. For example, a fuzzer could choose the value `CAD` and then mutate it to `CDF` by hitting the overloaded endpoint twice. We run each fuzzer on both the original specification and the enhanced specification.

We were unfortunately unable to directly compare to the approach in Kim. et. al [10]. We have filed an issue and have an ongoing discussion with the authors on the use of their artifact, and the paper does not directly ablate example generation. We did randomly sample 700 of our 13,346 mined API parameters (approximately 5%), finding that only 43 enumerated examples occur in the description. Thus, even if Kim et. al. [10] extracts examples with 100% accuracy, it could only do so for 6% of all API parameters.

3) Metrics: We use a combination of API fuzzing metrics and code coverage to evaluate the performance change of adding examples to each fuzzer.

Proportion of 2xx Requests: 2xx requests represent successful invocations of API endpoints, i.e., requests that yielded an HTTP response code between 200–299[7]. These requests are saved for further fuzzing; thus having more 2xx requests means that the fuzzer is capable of testing functionality beyond simple input validation.

Proportion of 4xx Requests: 4xx requests represent poorly formatted invocations, where the fuzzer invokes the API incorrectly. Ideally, a fuzzer should make fewer 4xx requests, as these are not testing deep functionality and wasting the fuzzing time budget.

Proportion of 5xx Requests: 5xx requests represent internal server errors. The goal of fuzzing is to catch such errors, thus more 5xx requests represent a successful fuzzing effort.

Branch Coverage: The goal of fuzzing efforts is to automatically test as much of the API as possible. Coverage is important, as higher code coverage indicates the fuzzer is testing a larger proportion of the API. We report branch coverage achieved by each fuzzer, as well as averaged across all four.

4) Results: Table II shows results across all fuzzers. Besides EvoMaster, all fuzzers exhibit a similar trend: examples lead to more 2xx requests (around 3%), fewer 4xx requests (around 3%), and around the same 5xx requests. This means that our example generation approach can seamlessly integrate with fuzzers to improve API testing, by better seeding fuzzers with realistic parameter examples that the fuzzers can use to invoke APIs.

[7] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

TABLE II: First three columns: API performance results for RESTest/ARTE, EvoMaster, MoREST, and RestTestGen across all 7 APIs. The proportion of 2xx requests goes up, 4xx goes down and 5xx slightly increases with enhanced examples. EvoMaster is the one exception, with 2xx request proportions decreasing. Last column: Coverage results. Coverage universally increases across all fuzzers with our enhancements.

	Freq. of 2xx			Freq. of 4xx			Freq. of 5xx			Branch Cov.		
Tool Name	**Base**	**Enhanced**	**Diff**	**Base**	**Enhanced**	**Diff**	**Base**	**Enhanced**	**Diff**	**Base**	**Enhanced**	**Diff**
RESTest/ARTE	0.28	0.31	+11%	0.57	0.54	-5%	0.10	0.10	+0%	4.0	6.2	+57%
MoREST	0.01	0.04	+300%	0.83	0.81	-2%	0.10	0.10	+0%	3.2	17.5	+447%
EvoMaster	0.29	0.24	-17%	0.58	0.62	+7%	0.08	0.07	-13%	6.9	12.2	+77%
RestTestGen	0.21	0.26	+24%	0.63	0.59	-6%	0.10	0.13	+30%	11.5	19.1	+66%
Average	0.20	0.21	+5%	0.65	0.64	-2%	0.10	0.10	+0%	6.4	13.8	+116%

Listing 4: An EvoMaster code snippet that explains why ICICL does not improve proportion of 2xx requests. EvoMaster adds a default value of ”EVOMASTER” to every enum, causing our generated examples to degrade API fuzzing performance.

```
if (schema.enum?.isNotEmpty() == true) {
    //Besides the defined values, add one to
        test robustness
    when (type) {
        "string" -> return EnumGene(name,
            (schema.enum as
            MutableList<String>).apply {
            add("EVOMASTER") })
    }
}
```

The reason EvoMaster does worse with ICICL is their addition of a default value of EVOMASTER to each enum list.[8] Listing 4 shows the code snippet responsible for the performance degradation. This affects our approach because we use path overloading to define an endpoint with our examples and a normal endpoint, allowing the fuzzer to perform standard mutation while leveraging our correct examples for better seeding. Due to this implementation detail, ICICL leads to a higher proportion of 404 requests for EvoMaster.

Branch coverage increases across all four fuzzers (from 6.4% to 13.8%). This means that in the time budget of one hour, more code is exercised. Although we do not show detailed results in the interest of space, method coverage also increases (from 11.9% to 17.8%, on average), meaning that more API enpoints are being hit as well.

B. API ChatBots

API chatbots take a natural-language utterance from a human and respond with the appropriate set of API calls. To accomplish this, API chatbots have intermediate tasks including recognizing which API a human wants to invoke (intent recognition) and filling in the values for each API parameter (slot filling). We evaluate how our examples affect the performance of these intermediate tasks on two benchmarks: MixATIS [31] and Schema Guided Dialog (SGD) [32].

1) Dataset: MixATIS [31] is a dialog benchmark developed to measure the ability of chatbots to deal with mixed-intent statements. Mixed-intent statements consist of multiple API invocations in the same natural language statement. The goal is to successfully produce the correct set of APIs to invoke and fill in slot values for each API parameter.

Listing 5: An example SeqATIS conversation and corresponding APIs. We use this model input and output to fine-tune an intent and slot filling chatbot on SeqATIS both with and without examples generated by ICICL.

```
Model Input: what is the name of the airport in new
    york and then what is the distance between new
    york airport and downtown atlanta

Model Output:
(1) API : "atis_airport",
    Parameters : [ city_name : philadelphia ]
(2) API : "atis_distance",
    Parameters : ...elided...
```

Listing 6: SGD model input. We experiment with removing the example conversation and replacing the example values with examples generated by ICICL.

```
Example Conversation: [example]
[user] what’s my balance? [system] in checking or
    savings?...
[slots] recipient_account_type=b of possible values
    a) checking b) savings

Model Input: [context]
[user] i’m paying some bills [system] which account?
     checking or savings?...
Model Output: [state]
account_type=b of possible values a) checking b)
    savings...
```

We use a version of the MixATIS dataset adapted for sequence to sequence (seq2seq) models called SeqATIS. SeqATIS encodes all conversations and corresponding slot values in text. Listing 5 shows an example of a mixed turn conversation and the corresponding APIs. The input to the model is a sequence of text and the output from the model is a list of all APIs to call (intent recognition) and the values to send each API (slot filling).

Similarly, the SGD benchmark [32] is a diverse collection of dialogue interactions. The benchmark covers multiple domains, including travel, services, and retail. Each conversation in the dataset is annotated with dialogue states and system actions.

We follow the methodology of Gupta et. al. [16] in constructing our dataset. Listing 6 shows an example of how

[8] https://github.com/rapesil/EvoMaster/blob/master/core/src/main/kotlin/org/evomaster/core/problem/rest/RestActionBuilderV3.kt

we (and Gupta et. al. [16]) fine-tune LLMs on the SGD dataset. We show the model an example conversation and then prompt it with the current conversation. The model is fine-tuned to produce a list of slot values (values for each API parameter in the API being invoked).

The SeqATIS and SGD benchmarks are not well-formatted OpenAPI specifications and lack attributes (e.g., operationId, endpoint path) required by ARTE, preventing a comparison against that technique here. ICICL is able to handle this case, allowing us to measure its impact on dialog tasks.

2) Approach: Since the SeqATIS dataset lacks descriptions, we use ChatGPT to generate descriptions for the 17 endpoints, and use these descriptions to generate examples with ICICL. These examples are then used as a prefix to each conversation when fine-tuning the LLM. For these experiments, we use FlanT5-XXL [33], as its instruction tuning enables it to generalize to new unseen tasks.

For our SGD evaluation, we investigate how our examples compare to the human-generated "gold" examples. We evaluate under the following settings: replacing the human-generated natural language conversation with just "gold" examples, replacing the conversation with examples generated by ICICL, and not providing any examples. For each of these settings, we fine-tune FlanT5-large. Our baseline numbers are comparable with Gupta et. al [16].

3) Metrics: We evaluate the performance of dialog systems that are fine-tuned on the SeqATIS in both the intent recognition and slot filling tasks. For SeqAtis we measure both intent and slot filling exact match and normalized scores. Normalized scores, unlike exact match, allow for partial correctness, for example if 1/3 slots in a natural-language utterance are matched correctly, the exact match slot score would be 0% for that example while the normalized slot matches score would be 33%. Overall score measures correctness for both intent and slot matching for each API invoked in an utterance, and thus, is harder than either task individually.

For SGD we only evaluate slot filling performance, as intents are provided as part of the input. We also report normalized slot match score, which is the proportion of matching slots across all data, rather than the stricter exact match score which requires all the slots in an example to be correct in order to be counted as correct.

4) Results: Table III shows results adding both examples and description as part of fine-tuning Flan-T5-XL. Performance on both slot filling and intent detection improves substantially. Both examples and description improve overall performance, with normalized intent and overall score going up with examples. Slot matches stays the same with examples, but description adds a significant performance improvement. Adding both examples and description improves performance between 3-5%.

TABLE III: SeqATIS intent recognition and slot filling scores. Original corresponds to the orignal specification, w/examples corresponds to adding examples generated by ICICL and w/both corresponds to adding both API examples and API descriptions. We find that examples generated by ICICL along with descriptions generated by ChatGPT improve performance on both intent recognition and slot filling for SeqATIS.

Metric	Original	w/examples	w/both
normalized intent	0.92	0.94	**0.95**
normalized overall	0.21	0.22	**0.24**
normalized slot matches	0.76	0.76	**0.81**
exact match intent	0.90	0.92	**0.94**
exact slot matches	0.74	0.74	**0.80**
exact match overall	0.20	0.21	**0.23**

TABLE IV: SGD slot filling scores. We find that replacing human generated examples with synthetic examples generated by ICICL results in little degradation in overall performance (1-2% across all metrics).

Setting	Exact slot	Normalized slot
hand-written examples	**0.74**	**0.95**
no description	0.73	**0.95**
ICICL	0.71	0.94
no examples	0.13	0.61

Table IV shows the performance degradation from replacing an example conversation with only example slot values and drop in performance from replacing example values with example values generated by ICICL. The core difference is for baseline and no-description settings, one would need to manually curate these example conversations or slot values, requiring developer time and effort. Using ICICL, we automatically generate slot examples for all parameters, with no human effort. The performance difference is not significant either, with an exact match slot score of 0.71 and normalized slot match score of 0.94 using ICICL, while using gold examples has a exact match score of 0.73. Without examples, the performance on slot filling drops significantly, with an exact match score of 0.13 and normalized slot match score of 0.61. This shows that examples (even if they are synthetic) are essential for slot filling performance. For a 2% degradation in exact match slot score and 1% degradation in normalized slot match score, a developer could theoretically use a fully synthetic approach to generating examples.

C. Human API Understanding

We conduct an exploratory study that aims to evaluate how useful the generated examples are for developers to understand a given OpenAPI Specification.

1) Dataset: We select four API endpoints from our software testing dataset as specifications to evaluate human understanding of APIs. We choose APIs of varying difficulty. Two are "easy", with a single parameter that is required, while two are "moderate", with six parameters, two of which are required. We modify specifications from the original to meet these requirements, for example marking parameters as required. We further modify specifications by removing any existing examples, any examples from the descriptions, and any default examples such that only generated examples are available to the participant. The modified endpoints used for the study are available as part of the replication kit.

2) Approach: We recruit six participants from a large technology company. Due to the cost of recruiting expert

Listing 7: Genome-Nexus human study specification (with examples). We ask participants to write a natural language summary and to generate four valid cURL invocations of this API to assess API understanding.

```
https://www.genomenexus.org/pfam/domain/{
    pfamAccession}:
    get:
        operationId:
            fetchPfamDomainsByAccessionGET
        parameters:
        -   name: pfamAccession
            description: A PFAM domain accession
                ID.
            required: true
            schema:
                type: string
                examples:
                -   PF00001
                -   PF00045
                -   PF00069
```

participants, we had a limited sample size, however this is in line with Crasswell [34], which states small samples can be suitable for certain exploratory studies. Each participant is self-reported to be experienced in using OpenAPI Specifications. The study design is within-subjects [34], where each participant was exposed to two conditions: using a specification with and without generated examples. Each participant performed four tasks total (on four different APIs) across the two conditions, shuffled to avoid cross-condition learning effects. Each condition has a pair of easy and moderate difficulty endpoints with the same two tasks per endpoint: write a summary of what the endpoint does in natural language and create four example invocations of the endpoint. Listing 7 shows an example of an easy specification. There is only one parameter marked as required. This specification has three examples generated by ICICL. We shuffle the order of conditions and endpoints used per participant such that half of the participants start without examples and that endpoints are evenly distributed between conditions. As we request example invocations using cURL (a simple command-line tool for transferring data with URLs),[9] we also provide a brief tutorial for writing cURL invocations that is available for each participant. At the end of the four tasks, we ask participants to provide open-ended feedback for how they understand API specifications, examples in specifications, and the study in general. Two researchers administered each study and recorded detailed notes. Although we did not perform a formal qualitative analysis for this exploratory study, we provide anecdotes from participants and notes to give some nuance to the quantitative results.

3) Metrics: In addition to qualitative metrics, we compute both accuracy and task completion time. For accuracy, we evaluate if the participant's generated cURL queries would return a 200 status code. We consider cURL queries where the parameters are correct values but the syntax is incorrect as correct for the purposes of our study. However, if any of the API parameters has any values that are incorrect, we mark the entire query as incorrect. We also measure the time to complete each task for both settings. We report both the average time to complete each task and accuracy for each combination of task and setting.

TABLE V: Summary of tasks and settings. We find that providing developers with examples allows for faster task completion rates, with no effect on overall correctness.

API	Examples	Time (s)	Correct	Attempted
Task 1	Yes	414	9/12	10/12
	No	712	6/12	11/12
Task 2	Yes	168	6/12	12/12
	No	289	6/12	11/12
Task 3	Yes	664	7/12	9/12
	No	708	11/12	12/12
Task 4	Yes	620	5/8	8/8
	No	468	4/12	10/12

4) Results: Table V shows the quantitative results from our exploratory study of six experts. The average task completion time decreased when generated examples were present (around 50%). This aligns with multiple participants' qualitative statements, where participants report using examples when understanding the meaning of an API. One surprising case was the decrease in accuracy for Task 3 from 11/12 to 7/12 when examples were present. This is likely due to the particpant attempting fewer of the problems, along with some of the generated examples being incorrect and blindly copied.

Interestingly, for more difficult tasks, the time effect decreased, with 1/2 intermediate tasks taking longer with examples than without examples. This could be because of bloated specifications; one participant stated "But I generally tell my developers, get your parameter examples out of my code because you're just adding too much junk." Too many examples can pollute the specification, making it harder to read and comprehend for a human. Accuracy remains relatively stable across both conditions, with participants eventually figuring out how to solve the task with and without API examples (just taking more time to do so).

In tasks where no examples were provided, participants repeatedly complained about the overall quality of specifications. One participant stated, "I think it's a pretty bad API", while another stated "Sample IDs would be useful." Without examples, developers struggle to understand and use APIs in their workflows.

In the majority of cases, participants seemed to appreciate having generated examples over not having any examples. Multiple participants reported improved guessing based off of examples, with one stating "I can only guess based on the examples" and another stating "At least this one is good, it has examples." One limitation of these LLM-generated examples is that participants tended to blindly trust the API examples provided to them, with the majority of participants in our study at some point copying an example into the cURL request. Since LLM-generated examples are not guaranteed to be fully accurate, blindly trusting these examples as oracles can lead to

[9] https://curl.se/

an increased proportion of badly formed or incorrect requests.

VI. Discussion

We discusses the broader impacts of our work in the context of software documentation, chat bots, and fuzz testing.

IcIcl complements existing automated documentation approaches: generated API examples can be used to enhance existing API documentation. IcIcl can generate examples for any API, making IcIcl more useful for company-specific APIs. IcIcl works as APIs evolve, while prior approaches would rely on the changed API existing on the internet or values being enumerated in the description.

IcIcl improves the performance of chat-bots by providing LLM-generated examples of internal APIs, giving them context to better understand the inputs and outputs of each API. This leads to both improved intent recognition (which API to call) and slot filling (what parameter values to pass). This also applies to customer-facing APIs, with LLM-generated examples both improving documentation and automated chat-bot services. Our evaluation of IcIcl on both MixATIS and SGD datasets shows that our examples improve API performance across both intent recognition and slot filling. Interestingly, we find that LLM-generated examples achieve comparable performance to human-generated examples, with only a 2% drop on slot filling exact match on SGD.

Another important area for companies is testing [35], [36]. Even though developers write unit and integration tests, many corporate services continue to be under-tested. Fuzzing helps find bugs in APIs [37], [38], and has been widely adopted by companies such as Google [39]. IcIcl improves the performance of fuzzers by providing realistic examples of API parameters, which can be used to seed these fuzzers, scaling to large APIs such as YouTube. Examples generated using IcIcl improve both API coverage and the proportion of 2xx requests. Practically, this means that companies can add IcIcl to their workflows and explore deeper code paths within the same fuzzing budget.

VII. Limitations and Threats

Internal Threats: An internal threat to validity is our implementation of IcIcl. To mitigate this, we used well-known programming libraries to construct prompts and publicly release our code. Another concern is that the LLMs we used may have seen API parameters at pretraining time. We attempt to mitigate by using Falcon 40B, a large and new model (which generally have lower leakage rates than older and smaller models) [40].

External Threats: An external threat is that we do not evaluate new bugs found by our technique for our fuzzing evaluation. Accurately bucketing fuzz crashes is an open challenge and we lack access to source code for all tested endpoints (instead we follow the metrics in NLP2Rest [10] and the original ARTE paper [9]). Finally, the small sample size of our exploratory study of six expert participants could limit the broader applicability of our findings. As such, the outcomes from our exploratory study should be considered as preliminary and interpreted with caution until larger, more comprehensive studies can be conducted.

Construct Threats: A construct validity concern is our selection of evaluation metrics. We used metrics commonly employed in evaluating dialog systems, fuzzing, and the broader field of machine learning, including coverage and exact match. While widely accepted, these metrics may not fully capture the complexities of the dialog systems we are analyzing.

VIII. Related Work

OpenAPI Specification Enhancement: Several approaches aim to create components of OpenAPI specifications from natural language specifications [41], [42], [43]. ARTE [9] mines examples from knowledge bases for use in fuzz testing. Recent approaches [10], [44] use language models to generate requests and other OpenAPI fields. Unlike existing techniques, which require examples to occur in knowledge bases or OpenAPI descriptions, IcIcl works on all OpenAPI specifications, generating relevant parameter examples.

Leveraging OpenAPI Specifications: There has been extensive work leveraging OpenAPI specifications for a variety of downstream tasks including chatbots, intent recognition, and business workflows [11], [45], [12]. Multiple fuzzers [46], [47], [8] actively use refined specifications provided by OpenAPI to improve API testing. Unlike these prior works, which focused on a single downstream task each, we evaluate our approach across several downstream tasks.

Large Language Models: Large language models (LLMs) can perform well across many tasks when prompted with instructions and examples [48], [49]. LLMs such as ChatGPT [14], GPT-4 [15] and LLAMA [49] perform well on a large range of natural language [50], [51] and code [52], [53], [54] and testing [55] tasks with minimal examples.

We leverage these LLMs to automatically improve OpenAPI specifications. We chose to not use extremely large models such as GPT-4 or Copilot due to company data leakage concerns and their high operating costs. We did experiment with several smaller models than Falcon-40B. While the results with these models were worse in absolute terms, IcIcl yielded similar improvement on these models in relative terms.

IX. Conclusion

We developed IcIcl, an LLM-based prompting approach to generate examples for OpenAPI parameters, performing both an intrinsic and multiple extrinsic evaluations. Intrinsically, IcIcl is capable of generating both correct and diverse examples. Extrinsically, IcIcl's examples can be applied to a wide variety of downstream tasks, including software testing and dialog systems. Our generated examples significantly improve performance in these tasks, increasing branch coverage by 116%, dialog intent recognition by 3%, and dialog slot filling by 5%, compared to the original specifications. Overall, IcIcl leverages the strong prior of LLMs to generate OpenAPI examples for a wide variety of API parameters that were not possible in prior work, improving the downstream performance of several tasks that use API examples.

References

[1] D. Jacobson, G. Brail, and D. Woods, "Apis: A strategy guide," 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:167782407

[2] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[3] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000, aAI9980887.

[4] "Openapi specification," Web page, 2023, accessed May 2024. [Online]. Available: https://spec.openapis.org/oas/v3.0.3

[5] S. Karlsson, A. Causevic, and D. Sundmark, "Quickrest: Property-based test generation of openapi-described restful apis," *CoRR*, vol. abs/1912.09686, 2019. [Online]. Available: http://arxiv.org/abs/1912.09686

[6] Z. Lei, Y. Chen, Y. Yang, M. Xia, and Z. Qi, "Bootstrapping automated testing for restful web services," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, p. 1561–1579, jun 2022. [Online]. Available: https://doi.org/10.1109/TSE.2022.3182663

[7] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao, "Morest: Model-based restful api testing with execution feedback," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1406–1417. [Online]. Available: https://doi.org/10.1145/3510003.3510133

[8] A. Arcuri, "Evomaster: Evolutionary multi-context automated system test generation," *CoRR*, vol. abs/1901.04472, 2019. [Online]. Available: http://arxiv.org/abs/1901.04472

[9] J. C. Alonso, A. Martin-Lopez, S. Segura, J. M. García, and A. Ruiz-Cortés, "Arte: Automated generation of realistic test inputs for web apis," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 348–363, 2023.

[10] M. Kim, D. Corradini, S. Sinha, A. Orso, M. Pasqua, R. Tzoref-Brill, and M. Ceccato, "Enhancing rest api testing with nlp techniques," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1232–1243. [Online]. Available: https://doi.org/10.1145/3597926.3598131

[11] M. Vaziri, L. Mandel, A. Shinnar, J. Siméon, and M. Hirzel, "Generating chat bots from web API specifications," in *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2017, pp. 44–57. [Online]. Available: http://doi.acm.org/10.1145/3133850.3133864

[12] Y. Rizk, V. Isahagian, S. Boag, Y. Khazaeni, M. Unuvar, V. Muthusamy, and R. Khalaf, "A conversational digital assistant for intelligent process automation," in *International Conference on Business Process Management – Blockchain and Robotic Process Automation Forum*, 2020, pp. 85–100.

[13] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," 2023. [Online]. Available: https://arxiv.org/abs/2302.04761

[14] "ChatGPT," Nov. 2022, accessed May 2024. [Online]. Available: https://openai.com/blog/chatgpt/

[15] OpenAI, "Gpt-4 technical report," 2023.

[16] R. Gupta, H. Lee, J. Zhao, Y. Cao, A. Rastogi, and Y. Wu, "Show, don't tell: Demonstrations outperform descriptions for schema-guided task-oriented dialogue," in *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Seattle, United States: Association for Computational Linguistics, Jul. 2022, pp. 4541–4549. [Online]. Available: https://aclanthology.org/2022.naacl-main.336

[17] A. S. M. Venigalla, C. S. Lakkundi, V. Agrahari, and S. Chimalakonda, "Stackdoc - a stack overflow plug-in for novice programmers that integrates q&a with api examples," in *2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT)*, vol. 2161-377X, 2019, pp. 247–251.

[18] M. Ichinco, W. Y. Hnin, and C. L. Kelleher, "Suggesting api usage to novice programmers with the example guru," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1105–1117. [Online]. Available: https://doi.org/10.1145/3025453.3025827

[19] T. Wanwarang, N. P. Borges, L. Bettscheider, and A. Zeller, "Testing apps with real-world inputs," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, ser. AST '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–10. [Online]. Available: https://doi.org/10.1145/3387903.3389310

[20] O. Rubin, J. Herzig, and J. Berant, "Learning to retrieve prompts for in-context learning," *CoRR*, vol. abs/2112.08633, 2021. [Online]. Available: https://arxiv.org/abs/2112.08633

[21] X. Dong, Z. Yu, W. Cao, Y. Shi, and Q. Ma, "A survey on ensemble learning," *Frontiers of Computer Science*, vol. 14, pp. 241–258, 2020.

[22] "APIS.guru," accessed May 2024. [Online]. Available: https://apis.guru/

[23] E. Wittern, J. Laredo, M. Vukovic, V. Muthusamy, and A. Slominski, "A graph-based data model for api ecosystem insights," in *2014 IEEE International Conference on Web Services*, 2014, pp. 41–48.

[24] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Found. Trends Inf. Retr.*, vol. 3, no. 4, p. 333–389, apr 2009. [Online]. Available: https://doi.org/10.1561/1500000019

[25] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, "Improving few-shot prompts with relevant static analysis products," in *International Conference on Software Engineering (ICSE)*, May 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639183

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018, cite arxiv:1810.04805Comment: 13 pages. [Online]. Available: http://arxiv.org/abs/1810.04805

[27] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1159–1170. [Online]. Available: https://doi.org/10.1145/3377811.3380362

[28] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "How hard does mutation analysis have to be, anyway?" in *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. IEEE Computer Society, 2015, pp. 216–227. [Online]. Available: https://doi.org/10.1109/ISSRE.2015.7381815

[29] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Restest: Automated black-box testing of restful web apis," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 682–685. [Online]. Available: https://doi.org/10.1145/3460319.3469082

[30] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: Automated black-box testing of restful apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 142–152.

[31] L. Qin, X. Xu, W. Che, and T. Liu, "AGIF: An adaptive graph-interactive framework for joint multiple intent detection and slot filling," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1807–1816. [Online]. Available: https://www.aclweb.org/anthology/2020.findings-emnlp.163

[32] A. Rastogi, X. Zang, S. Sunkara, R. Gupta, and P. Khaitan, "Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset," *CoRR*, vol. abs/1909.05855, 2019. [Online]. Available: http://arxiv.org/abs/1909.05855

[33] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, E. Li, X. Wang, M. Dehghani, S. Brahma *et al.*, "Scaling instruction-finetuned language models," *arXiv preprint arXiv:2210.11416*, 2022.

[34] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*, 4th ed. SAGE publications, 2013.

[35] G. Petrovic and M. Ivankovic, "State of mutation testing at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, F. Paulisch and J. Bosch, Eds. ACM, 2018, pp. 163–171. [Online]. Available: https://doi.org/10.1145/3183519.3183521

[36] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 91–100.

[37] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[38] K. Serebryany, "OSS-Fuzz - google's continuous fuzzing service for open source software." Vancouver, BC: USENIX Association, Aug. 2017.

[39] J. Metzman, L. Szekeres, L. M. R. Simon, R. T. Sprabery, and A. Arya, "Fuzzbench: An open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2021.

[40] D. Ramos, C. Mamede, K. Jain, P. Canelas, C. Gamboa, and C. L. Goues, "Are large language models memorizing bug benchmarks?" Nov. 2024. [Online]. Available: https://arxiv.org/abs/2411.13323

[41] S. Huo, K. Mukherjee, J. Bandlamudi, V. Isahagian, V. Muthusamy, and Y. Rizk, "Natural language sentence generation from API specifications," 2022.

[42] J. Yang, E. Wittern, A. T. T. Ying, J. Dolby, and L. Tan, "Towards extracting web API specifications from documentation," in *Conference on Mining Software Repositories (MSR)*, 2018, pp. 454–464.

[43] G. Baudart, P. Kirchner, M. Hirzel, and K. Kate, "Mining documentation to extract hyperparameter schemas," in *ICML Workshop on Automated Machine Learning (AutoML@ICML)*, Jul. 2020. [Online]. Available: https://arxiv.org/abs/2006.16984

[44] A. Decrop, G. Perrouin, M. Papadakis, X. Devroey, and P.-Y. Schobbens, "You can REST now: Automated specification inference and black-box testing of RESTful APIs with large language models," Feb. 2024. [Online]. Available: https://arxiv.org/abs/2402.05102

[45] 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.

[46] P. Godefroid, B.-Y. Huang, and M. Polishchuk, "Intelligent REST API data fuzzing," in *Symposium on the Foundations of Software Engineering (FSE)*, 2020, p. 725–736.

[47] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *International Conference on Software Engineering (ICSE)*, May 2019, pp. 748–758.

[48] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[49] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[50] T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee, K. Toutanova, L. Jones, M. Kelcey, M.-W. Chang, A. M. Dai, J. Uszkoreit, Q. Le, and S. Petrov, "Natural questions: A benchmark for question answering research," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 452–466, 2019. [Online]. Available: https://aclanthology.org/Q19-1026

[51] M. Joshi, E. Choi, D. Weld, and L. Zettlemoyer, "TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 1601–1611. [Online]. Available: https://aclanthology.org/P17-1147

[52] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[53] G. Verbruggen, V. Le, and S. Gulwani, "Semantic programming by example with pre-trained models," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485477

[54] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *International Conference on Software Engineering (ICSE)*, May 2023, pp. 2450–2462. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00205

[55] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *International Conference on Software Testing, Verification and Validation (ICST)*, May 2012, pp. 260–269. [Online]. Available: https://doi.org/10.1109/ICST.2012.106