DeepREST: Automated Test Case Generation for REST APIs Exploiting Deep Reinforcement Learning

Davide Corradini
davide.corradini@univr.it
University of Verona
Verona, Italy

Zeno Montolli
zeno.montolli@univr.it
University of Verona
Verona, Italy

Michele Pasqua
michele.pasqua@univr.it
University of Verona
Verona, Italy

Mariano Ceccato
mariano.ceccato@univr.it
University of Verona
Verona, Italy

ABSTRACT

Automatically crafting test scenarios for REST APIs helps deliver more reliable and trustworthy web-oriented systems. However, current black-box testing approaches rely heavily on the information available in the API's formal documentation, i.e., the Open API Specification (OAS for short). While useful, the OAS mostly covers syntactic aspects of the API (e.g., producer-consumer relations between operations, input value properties, and additional constraints in natural language), and it lacks a deeper understanding of the API business logic. Missing semantics include implicit ordering (logic dependency) between operations and implicit input-value constraints. These limitations hinder the ability of black-box testing tools to generate truly effective test cases automatically.

This paper introduces DeepREST, a novel black-box approach for automatically testing REST APIs. It leverages *deep reinforcement learning* to uncover implicit API constraints, that is, constraints hidden from API documentation. Curiosity-driven learning guides an agent in the *exploration* of the API and learns an effective order to test its operations. This helps identify which operations to test first to take the API in a testable state and avoid failing API interactions later. At the same time, *experience* gained on successful API interactions is leveraged to drive accurate input data generation (i.e., what parameters to use and how to pick their values). Additionally, DeepREST alternates exploration with *exploitation* by mutating successful API interactions to improve test coverage and collect further experience.

Our empirical validation suggests that the proposed approach is very effective in achieving high test coverage and fault detection and superior to a state-of-the-art baseline.

KEYWORDS

REST API Testing, Deep Reinforcement Learning, Automated Black-box Testing

ACM Reference Format:
Davide Corradini, Zeno Montolli, Michele Pasqua, and Mariano Ceccato. 2024. DeepREST: Automated Test Case Generation for REST APIs Exploiting Deep Reinforcement Learning. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695511

1 INTRODUCTION

With the exponential growth of web applications and the increasing complexity of software systems, the demand for efficient and reliable testing methodologies has become paramount. Among the various forms of testing, the black-box testing of *REpresentational State Transfer* (REST) APIs has garnered significant attention due to their widespread adoption in modern web architectures.

In literature, several black-box REST API testing approaches have been proposed [1], mostly relying on the information available in the *OpenAPI Specification* (OAS), that is the API formal documentation of the system under test. However, it has been observed that this information often falls short of addressing the intricate challenges associated with REST API testing. Specifically, two primary challenges arise in black-box REST API testing: *(i)* the selection of an effective ordering of API operations to test; and *(ii)* the generation or retrieval of valid input values for such operations.

In contrast to web or mobile application testing, where the next test interaction can often be deduced from the application's context (e.g., available links or widgets in the graphical user interface), REST APIs expose no such explicit context. Indeed, API operations can be, in principle, called at any time, and the OAS usually does not encode any information about the prioritization of operation invocations. This may lead to failures in crafting test cases, not just due to incorrect generation of operation input values but due to calling an operation that is not (yet) ready to be called, given the current API state.

As a simple example, consider the case of an e-commerce service. When the shopping cart is empty, the checkout operation is supposed to fail independently of the input values provided to the operation simply because the cart is not ready for checkout. This suggests that an operation adding items to the cart must be called first to make the cart available for checkout. Some state-of-the-art black-box testing tools [2–4] alleviate the problem of operation ordering by considering producer-consumer relations

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695511

between API operations. This yields an operation ordering purely based on explicit data dependencies derivable from the OAS (that is, an operation is tested before another when the latter requires as input a resource provided as output by the former). Even if effective in simple cases, such an approach overlooks *implicit* operation dependencies caused by *logic constraints* rather than *data constraints*, as in the case of the checkout example above. Conversely, *spurious constraints* could be incorrectly inferred from the OAS, driving testing tools towards inappropriate ordering of operation sequences. Indeed, an OAS does not fully encode the API business logic, resulting in API constraints that cannot be statically inferred from it.

In addition to the correct operation ordering, another crucial goal when testing REST APIs is adopting valid input values to call operations. This task is particularly hard due to the extensive value exploration required within a vast input space. Valid input values can encompass various combinations of data, which may need to satisfy inter-parameter dependencies and value conditions. Some input constraints are documented in the OAS and exploited by testing tools [2, 3, 5–7] to craft correct HTTP requests. However, similarly to the operations dependency case, some implicit input constraints may not be syntactically retrieved from the OAS, decreasing the chances of crafting successful test cases. For instance, an operation adding products to the cart may fail if a discounted product is added in a quantity less than the minimum. Such constraint is known to the API developer but unlikely to be found in the OAS and, consequently, hidden from black-box testing tools. Moreover, smartly picked input values might bring a REST API to a new state that is interesting to test because it could expose new and more complex logic defects.

With the aim of automatically generating effective test cases for REST APIs, we propose DeepREST, a novel approach based on *deep reinforcement learning* to learn API constraints during testing, potentially even those not documented in the OAS (hence hidden from black-box tools). DeepREST trains an intelligent agent to autonomously learn and optimize a strategy to test a REST API in a black-box fashion. Deep reinforcement learning is leveraged to guide an agent in the *exploration* of several API states, which is positively rewarded when discovering an effective order in which to test API operations. At the same time, DeepREST also learns the most effective strategies to generate input values among the available strategies. Typical strategies are random generation, dictionary lookup, and reusing examples from the OAS. Such selection is based on the *experience* gained during previous successful interactions. Finally, DeepREST also performs *exploitation*, by mutating successful interactions and improve test coverage, fault detection, and collect even further experience.

The contribution of this paper can be summarized as follows:

- The first approach leveraging deep reinforcement learning to automatically learn an effective *testing order* for operations in a REST API;
- A novel reinforcement learning-based approach to select the most effective *input value generation strategy* for operation input parameters;
- Empirical results demonstrating that the proposed approach is *effective* (coverage and faults detection) and *efficient* (number of requests) in testing REST APIs. Indeed, DeepREST achieves superior performance than state-of-the-art testing tools a set of case study APIs;
- An open-source tool implementing the approach, that can be found in the replication package [8] and on GitHub [9].

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: "Simple eComm"
  license:
    name: MIT
servers:
  - url: http://simple.ecommerce.io/v1
paths:
  /addProductToCart:
    post:
      summary: "Add product(s) to the cart"
      operationId: addProductToCart
      parameters:
        - name: productId
          in: query
          required: true
          schema:
            type: integer
            format: int64
        - name: quantity
          in: query
          schema:
            type: integer
            default: 1
            minimum: 1
            maximum: 100
      responses:
        '200':
          description: "Product(s) added"
```

```yaml
  /products/search:
    get:
      summary: "Search products by name"
      operationId: productSearch
      parameters:
        - name: keyword
          in: query
          required: true
          schema:
            type: string
      responses:
        '200':
          schema:
            type: array
            items:
              properties:
                productId:
                  type: integer
                  format: int64
                name:
                  type: string
                price:
                  type: number
                  format: float
  /checkout:
    post:
      summary: "Finalize purchase"
      operationId: checkout
      responses:
        '200':
          description: "Purchase completed"
```

Figure 1: OpenAPI specification excerpt for *Simple eComm*.

2 BACKGROUND

This section covers the background notions needed to understand our approach. It includes an introduction to REST APIs and OpenAPI specifications, automated REST API testing guided by data dependencies, and reinforcement learning.

2.1 REST APIs and OpenAPI Specifications

The REST (REpresentational State Transfer) architectural style [10] is nowadays the most common paradigm adopted in web API development. A RESTful API (or REST API) is a web API that adheres to such a paradigm, allowing web clients to access and manipulate resources and invoke remote routines by leveraging stateless operations over the HTTP protocol.

REST APIs provide a uniform interface to *Create*, *Read*, *Update*, and *Delete* (CRUD) resources, where an HTTP URI identifies a resource while CRUD operations are typically mapped to the HTTP methods `POST`, `GET`, `PUT` (or `PATCH`) and `DELETE`, respectively. Additionally, REST APIs can expose functionalities such as resource search, invocation of remote routines, and authentication mechanisms. Upon receiving and processing an HTTP request that exercises a specific API operation, the REST API returns an HTTP response with the outcome of the request, called status code (e.g., 2XX for a success; 4XX for a client-side error; or 5XX for a server-side error), and, possibly, a payload.

As an example, consider *Simple eComm*, a REST API managing a simple e-commerce service. A possible HTTP URI pointing to the search products functionality could be `/products/search`. In this case, the HTTP operation `GET␣/products/search` is used to search products by name, listing all products matching the search keyword provided as an HTTP input parameter. Conversely, the

HTTP operation `POST␣/addProductToCart` could be used to add a product to the shopping cart.

REST APIs are usually documented by using the OpenAPI[1] standard. According to such standard, an API is described by a structured file (either YAML or JSON), called *OpenAPI Specification* (OAS), that indicates how to reach the API using a URI, which authentication schema is adopted, and the details of the available API operations: the input parameters (and their schema) to be used in requests and the schema of responses.

Figure 1 contains an excerpt of the OAS for *Simple eComm*. After an initial header that specifies versions, licenses, and the base URL of the REST API, an OpenAPI specification contains the list of available URL paths. In the example, we have, among others, the two paths `/products/search` and `/addProductToCart`.

A REST API *operation* is a pair of path and HTTP method, usually identified by an operation identifier. For instance, the method `GET` in `/products/search` refers to the operation identified as `productSearch`, where the search string is given as query parameter `keyword`. Similarly, the method `POST` in `/addProductToCart` refers to the operation named `addProductToCart`, where the product to add and quantity of the added product are given as query parameters `productId` and `quantity`, respectively.

Request input and output are associated with a schema that specifies their type and, optionally, a set of constraints on values (e.g., a minimum or a maximum value for numeric parameters as in the case of `quantity`). Types can be atomic (e.g., integers and strings) or structured (i.e., compound objects). For instance, the parameter `keyword` of `/products/search` is of type `string`, while the response to the corresponding `GET` operation is expected to be an array of product objects, whose schema is also provided in the specification.

2.2 Data Dependency-Based REST API Testing

Testing strategies of black-box approaches are typically based only on the information contained in OpenAPI specifications. So, to select an effective ordering of operations to test, state-of-the-art tools [2–4] purely base their decisions on data dependencies among the documented operations.

For instance, RESTler [2] infers data dependencies (in the form of producer-consumer relations) between the operations documented in the OAS. Then, by leveraging a search-based algorithm, it extensively generates sequences of HTTP requests conforming to the inferred dependencies. Instead, RestTestGen [3] computes the *Operation Dependency Graph* (ODG), a graph encoding data dependencies (again, producer-consumer relations) among operations available in the OAS. Finally, Morest [4] exploits the *Property Graph*, which captures OAS-induced relations between API operations, to prioritize operation testing order.

Referring to the *Simple eComm* example of Figure 1, the operation `addProductToCart` depends on the operation `productSearch`, which provides a list of valid products, since the output of the latter can be used as input for the former. Indeed, to test the operation `addProductToCart`, a valid `productId` value is needed that is unlikely to be guessed. Hence, the operation `productSearch` should be tested earlier in order to fetch a valid value for `productId`.

Data dependencies are statically inferred from the OAS by matching parameter names and schemas, giving higher priority to operations with satisfied data dependencies.

2.3 Reinforcement Learning

Reinforcement learning is a paradigm of machine learning in which an agent learns to make decisions by interacting with an environment. After taking action, an agent receives feedback in the form of rewards or penalties depending on the effect of its actions and adjusts its strategy over time to maximize cumulative rewards. The learning process involves discovering an optimal policy that guides decision-making.

Multi-Armed Bandit Problem. The multi-armed bandit problem is a decision-making scenario where an agent is confronted with a problem, where either exploration or exploitation should be aimed at the same time. This problem is named after its origin, where a set of slot machines is available to a player with a limited playing budget. Each slot machine has an unknown probability distribution and amount of winning, so the player has to spend money to *explore* the machines to find the best to play with but, at the same time, money should be invested to *exploit* best machines to gain profit. The challenge lies in the exploration vs. exploitation trade-off: the agent must strike a balance between trying different actions to uncover their reward potentials and exploiting the current best-known action for immediate gains. This dilemma reflects the tension between gaining more information and making optimal decisions based on existing knowledge.

A class of solutions to the multi-armed bandit problem, known as *probability matching strategies*, is based on the idea that the probability of choosing a solution should be equal to the probability for that solution to be optimal, according to the collected experience.

Deep Reinforcement Learning. Deep reinforcement learning combines reinforcement learning with deep neural networks, using the latter to represent the agent's policy or value function. The learning process revolves around the agent interacting with an environment, where it observes a state s_t, takes an action a_t, receives a reward r_t, and transitions to a new state s_{t+1}. The agent's goal is to learn a policy that maximizes the cumulative reward over time.

The *Proximal Policy Optimization* (PPO) algorithm [11] is a technique used in deep reinforcement learning to train an agent's decision-making abilities. PPO aims to improve an agent's policy through trial and error, balancing the policy update towards better performance while keeping it similar to the previous policy. This balance helps the agent learn effectively while maintaining stability.

3 MOTIVATING EXAMPLE

Let us consider the OAS of *Simple eComm* reported in Figure 1. In the excerpt, three operations are defined: `productSearch`, which searches for products by name; `addProductToCart`, which adds a product to the shopping cart; and `checkout`, which finalizes the purchase of the current shopping cart. Even if apparently simple, by using *Simple eComm* we can highlight three non-trivial challenges posed by REST APIs, that are only partially addressed by state-of-the-art testing approaches.

[1] https://www.openapis.org/

Challenge 1: Correct ordering of operations to test. To successfully test the `checkout` operation, we need a non-empty shopping cart, and this requires: *(i)* searching for existing products; *(ii)* adding them to the shopping cart; and *(iii)* performing the checkout.

State-of-the-art testing approaches based on data dependencies infer the ordering on which operations are tested by exploiting producer-consumer relations. The (explicit) data dependency between `productSearch`, which returns a `productId` as output, and `addProductToCart`, which requires a `productId` as input, would suggest testing `addProductToCart` after `productSearch`, using the `productId` returned by the latter. However, such approaches would overlook the business logic constraint that only valid and non-empty carts can be finalized. This results from an *implicit dependency* between `addProductToCart` and `checkout`, where the former operation takes the state where it can be purchased, and the latter operation finalizes the purchase. This dependency cannot be inferred just from the OAS of *Simple eComm* in Figure 1, but it could be learned by testing the API. We advocate for the need to overcome the limitations of current approaches and enhance automated testing so that implicit dependencies are learned on successfully tested scenarios, yielding an effective ordering of operations to test.

Challenge 2: Appropriate input data selection. The second limitation is in how state-of-the-art approaches decide input data. Typical strategies are either random generation according to the constraints in the API specification, sourcing values from it (e.g., default, enums, and example values), or reusing values observed on previous interactions (from HTTP responses). However, the strategy to be used is typically chosen randomly among those available, regardless of the role of the input parameter. For instance, the value of `productId` parameter in the `addProductToCart` operation may be wisely chosen by picking one of those values that have been observed in previous responses (e.g., to `productSearch`), rather than with random generation. Conversely, it is more effective to use a random query string to test `productSearch`, rather than previously observed item values. We advocate that the decision about the strategy to generate input data should exploit the *experience* on past successful interactions.

Challenge 3: Balance between exploration and exploitation. Lastly, a notable challenge is how to balance *exploration* of the REST API to test new operations and *exploitation* of already tested operations for increasing coverage by testing them with diverse inputs. In fact, thorough and in-depth testing of individual operations could yield two distinct benefits: *(i)* the acquisition of additional valid data from API responses and *(ii)* higher coverage of API source code.

In the context of the *Simple eComm* example, intensifying the testing on `searchProduct`, by trying different search keywords, would increase the chances of formulating valid queries and retrieving more and more shopping products. Additionally, having multiple valid `productId` values would facilitate the testing of the subsequent operation `addProductToCart` and, consequently, of the operation `checkout`.

4 APPROACH

In this section we present DeepREST, our approach conceived to address the challenges identified in the previous section. An overview of the architecture of DeepREST is illustrated in Figure 2, depicting an iterative process that applies three main components to craft REST API test cases, ending up in actual HTTP interactions. Such an iterative process is repeated until all API operations have been extensively tested or the allocated testing budget expires.

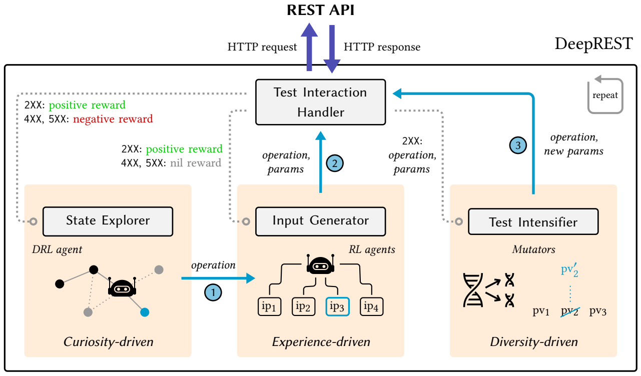

Figure 2: Approach overview.

The first component, the *Curiosity-Driven State Explorer*, is a deep reinforcement learning agent determining the most relevant next API operation to test based on the current state of the API. Employing a curiosity-driven strategy, this agent is positively rewarded when it reaches new API states, thereby *promoting exploration.* Its objective is to cover new states of the API beyond those reachable purely following data dependencies, discovering those that are reachable when implicit dependencies are exploited. Further details about this component are described in Subsection 4.1.

The second component, the *Experience-Driven Input Generator*, is responsible for supplying valid input data to operations. Internally, this component adopts a group of reinforcement learning agents, each dedicated to a distinct input parameter. Treating the parameter value selection as a multi-armed bandit problem, these agents *leverage experience* to make informed decisions. Experience is also exploited to decide whether to include non-mandatory parameters in requests and to determine arrays' size. Further details about this component are provided in Subsection 4.2.

The first two components collectively assemble an HTTP request for the API under test. Testing an API is learned by providing a positive reinforcement to correct interactions (HTTP responses with a status code 2XX) and a negative reinforcement to wrong interactions (HTTP responses with a status code 4XX or 5XX).

Upon successfully reaching a new API state and testing a new operation, a *test intensification phase* is initiated. The third component, the *Mutation-Based Test Intensifier*, performs deeper testing of the newly reached state, with the twofold objective of collecting additional test data from API responses and increasing test coverage. Intensification consists of applying mutation operators to a successful HTTP request to generate many new requests, each with small changes in parameter values, *promoting input diversity.* Further details about this component are described in Subsection 4.3.

4.1 Curiosity-Driven State Exploration

To comprehensively inspect the behavior of the API under test, it is crucial to invoke API operations in a clever order. As pointed out in Section 3, if the order in which operations are executed is based on data dependencies only, we might overlook potential *implicit dependencies* and, thus, fail to test some operations. Moreover, *spurious dependencies* could be inferred from the OAS, causing a testing approach to waste its testing budget on hopeless attempts. This limitation could result in a partial exploration of the API's state space, collecting few meaningful test data and causing low test coverage. On the flip side, it is not advisable to test all possible sequences of operations since the sheer volume of potential sequences is prohibitively vast. In addition, a substantial portion of these sequences lacks practical significance or coherence with the API business logic, making their testing unnecessary. Hence, a selective approach is crucial for efficient and effective testing.

Leveraging curiosity-driven deep reinforcement learning, we push toward exploring the API while learning an effective order of operations, without resorting to other techniques (such as static analysis of the specification to infer data dependencies) that might introduce bias into the learning process. We formulate the problem as an instance of reinforcement learning, with proper *state*, *actions*, *state transition*, and *reward* as follows.

State. In an ideal scenario, the *DRL state* should precisely reflect the internal state of the API under test. However, since we treat the API as a black box, its precise internal state is unknown. We then resort to an approximation of the API internal state, referred to as *state observation*, derived from the information we can retrieve from previous HTTP interactions with the API. For instance, a successful exercise of the `addProductToCart` operation inherently suggests that the shopping cart contains items. With this consideration, we represent the DRL state as a list of n integers, where n is the number of API operations. The value of the element at position i in the list represents the number of successes (HTTP interactions that returned a `2XX` status code) for the i-th operation in the OAS that occurred so far during testing.

As an example, consider the three operations of *Simple eComm*: `addProductToCart`, `productSearch`, and `checkout`. Then, the list [0 1 0] represents a state observation where the operation `productSearch` has been successfully tested (1 success), while the other two operations might have been attempted with no success (0 successes). In fact, successful operations are supposed to alter the (internal) state of an API, potentially enabling other operations to be called, while failed operations should have no side effects.

Considering that in our model state observation is multi-discrete, i.e., it is an array with discrete values, finite-state reinforcement learning techniques, such as Q-Learning [12] can not be applied, and *deep* reinforcement learning is needed.

Thanks to the generalizability of the approximation function in its neural network, a DRL agent can link similar states. On new states, it can take those actions that it learned on observed similar states [13]. Thus, even if the search space is very large, generalizability makes our approach not purely based on exploration and our agent can learn by generalizing its experience.

Actions. The agent actions are represented by the set of available API operations. In our example, the three actions are testing either `addProductToCart`, `productSearch`, or `checkout`.

State Transitions. The API transitions to a different (internal) state depending on the outcome of an operation's execution. If an operation execution resulted in success (HTTP status code `2XX`) the API is assumed to have reached a new state, potentially "enabling" operations that were infeasible to test in the previous state. In this case, we update the state observation by incrementing the success counter for the operation successfully tested. Conversely, in the event of failed interactions (HTTP status codes `4XX` and `5XX`), we assume that the API state did not change. In this case, also the state observation does not change.

Reward. To stimulate curiosity, a DRL agent usually receives a large positive reward whenever its action takes the environment to a new state never reached so far [14, 15]. In our context, new (internal) API states are approximated with new state observations, which are reached when the DRL agent is able to successfully test an operation. Here, we distinguish two cases: the operation is successfully tested for the first time (i.e., in the previous state observation its counter was 0); and the operation has been already successfully tested before (i.e., in the previous state observation its counter was greater than 0). In the first scenario, the agent receives a large positive reward (+1000), while in the second scenario, the agent receives a negative reward (−100). With this strategy, we stimulate curiosity and, at the same time, we do not overly discourage the agent from repeating certain operations (it can re-test the same operation up to 10 times before its overall reward becomes negative), which may be a prerequisite for successfully testing other operations.

Finally, if the API rejects the chosen operation (with a `4XX` or `5XX` status code), a slightly negative reward (−1) is assigned to the agent. The penalty is intentionally mild, recognizing that rejection can stem from various reasons. Indeed, the agent may have selected the correct operation, but value generation provided the wrong input data (the latter causing the failure). In such a case, the agent should not be discouraged from attempting again to test the operation with different input data. Recall that input generation is in charge of a different learning agent (that will be presented in Section 4.2), having a different reward strategy.

DRL Algorithm. To solve the aforementioned learning problem, we opted to employ the *Proximal Policy Optimization* algorithm (PPO) [11] since it is one of the most recent and advanced deep reinforcement learning algorithms [16] supporting vector state space with discrete values. In particular, we employed the PPO implementation coming with Stable Baselines 3 [17]. We customized the episode length to be equal to `ep_length` = 20 · `num_operations`, thus correlating it with the size API being tested. APIs exposing several operations will be granted more attempts in a single episode than smaller APIs. In a similar fashion, the maximum value for counters in the DRL state is set to 20, and in the case that an operation is successfully tested more than 20 times in an episode, the episode is truncated before its natural end, as the DRL agent could have stuck in choosing simple operations to test, rather than exploring others.

4.2 Experience-Driven Input Generation

Providing appropriate input data in API requests is crucial in automated black-box testing of REST APIs. Such input data comprises a collection of operation parameters that can take the form of simple values (such as strings, numbers, and booleans), arrays, or compound objects. In the process of generating input data, three challenges must be addressed: *(i)* deciding which parameters to include in the request among the optional ones; *(ii)* selecting the length of arrays; and, most notably, *(iii)* deciding the values to assign to primitive-type parameters (recursively on compound objects).

To address these challenges, we developed a novel approach based on cumulative experience inspired by the multi-armed bandit problem. A distinct agent is deployed for each parameter, responsible for making clever decisions about how that parameter should be used in requests, grounded in their accumulated experience. Agents' experience is initially empty, thus leading to initial purely random decisions. When successful input data is generated, agents receive a positive reward (+1) associated with the decisions they made which led to a successful interaction, accumulating valuable experience on how to succeed. Decisions are made following a probability-matching strategy, where the likelihood of making a specific decision corresponds to the observed statistical distribution estimated at that time, as follows. The probability of choosing d is equal to the ratio of the accumulated rewards for d (i.e., R_d) on the total rewards $\sum_i R_i$ accumulated so far, that is: $\mathbb{P}(d) = R_d / \sum_i R_i$.

For example, consider an agent deciding whether to include a parameter in a new request. Suppose that, based on the agent's accumulated experience, the parameter has been employed in 8 out of the 10 previously successful interactions (i.e., it has collected 8 reward points). The agent will incorporate that parameter in the new request with a probability of $\mathbb{P} = 0.8$. To prevent overfitting towards a single solution, however, agents might also make random decisions (with low probability, e.g., 0.1), still allowing the exploration of not yet tried configurations.

A notable aspect of our approach is that the experience garnered on a parameter in one operation is leveraged for making decisions on parameters with the same (or similar) name in other operations. This approach significantly reduces the learning time when testing new operations, promoting efficiency and knowledge reuse.

We now describe how we specifically address the three challenges mentioned at the beginning of the subsection.

Parameter Presence. For each optional parameter in an operation, the agent decides whether to include or exclude the parameter in the request. As already discussed, the probability of a parameter being included matches the statistical distribution estimated so far.

Array Length. Deciding the length of an array is inherently a hard problem, as arrays can theoretically have any size. We categorize array length into three classes to overcome the issue, yielding a bounded approach. In particular, *Class A* denotes empty arrays (the length is 0), *Class B* denotes one-element arrays, and *Class C* denotes arrays having at least 2 elements. Agents are rewarded positively (+1) if their selected size class leads to a successful request. When crafting new requests, agents will select the most suitable size class from the statistical distribution estimated so far. In case the *Class C* is selected, the actual size of the array is randomly chosen to be equal to or greater than 2 and compatible with the length constraints reported in the OAS.

Input Values. Our approach relies on a catalog of input generation strategies, or *sources of values*, that are wisely selected at testing time to retrieve the most appropriate value for parameters at a given point in testing time.

Input values could be taken from various sources, such as random generators, examples in the OAS, and dictionaries containing test data collected from previous HTTP interactions. However, the same source might not be equally effective for all the parameters. For instance, when dealing with resource identifiers, dictionaries are likely to be the optimal sources of valid values observed in the past rather than random values that are unlikely to be valid identifiers. Our approach accumulates and subsequently leverages knowledge about the most successful sources for each parameter.

When testing an operation, we keep track of the source that supplied the value for each parameter. Upon successful execution of an operation (status code 2XX), reinforcement learning agents obtain a positive reward (+1), promoting the reuse of the same source in the future. It is important to note that the same source is unlikely to provide the same, identical value for future requests (think of, for instance, the random generation source). However, sources employ the same strategy to select a value, which ensures further exploration of the API.

The value sources now available in DeepREST are the following.

Random Parameter value is randomly generated according to the constraints in the OAS.

Default Parameter value is the default value in the OAS.

Enum Parameter value is randomly taken from one of the valid enum values from the OAS.

Examples Parameter value is assigned with one of the example values from the OAS.

ResponseDictionary Parameter value is taken from a dictionary of API response values observed for this parameter in previous interactions. These values, coming directly from the API, are likely valid and, therefore, likely accepted in new requests.

LastResponseDictionary Same as the previous, but the last observed value is assigned to the parameter. This increases the likelihood of the value being valid. In the case of parameters acting as resource identifiers, it will help the generation of a chain of operations targeting the same resource.

RequestDictionary Parameter value is taken from a dictionary of values already used for this field, whose API requests obtained a successful status code. These values are likely valid since the API has accepted them in previous interactions.

LastRequestDictionary Same as the previous, but the last observed value is assigned to the parameter.

LargeLanguageModelDictionary A large language model is queried to supply values for parameters based on their names, the *context* of the endpoint, and the parameter description if provided in the OAS (further details in the next subsection).

Except for **LargeLanguageModelDictionary**, which is a contribution of the paper and detailed below, the previous value sources are inspired by literature [2, 3] (we adopted the implementation coming with the RestTestGen Framework [18]). For further customization, this catalog can be easily extended. The novel contribution of our approach is how to learn the most effective strategy from the catalog at each testing step rather than the catalog itself.

Large Language Model Dictionary. A new method for generating realistic parameter values in our implementation of DeepREST utilizes the inherent knowledge of large language models. We assume that a large language model can suggest realistic values for a parameter based on both its name (for instance, for a parameter named `title`, suggestions could include "The Odyssey" or "The Iliad") and the context of the specific API operation to which the parameter belongs (for example, within the operation `POST␣/person`, plausible values for a parameter named `title` could be, instead, "Mrs." or "Mr."). Furthermore, the natural language descriptions associated with the parameter and its API operation, sometimes found in the OAS, can further guide the language model in generating realistic values.

To implement this approach, before actually starting with the testing session, we ask an LLM to suggest plausible input values for all parameters in the OAS. This involves providing the LLM with the HTTP method and path of the API operation, the description of the API operation (if available in the OAS), the parameter name, its type, and its description from the OAS (if available). The language model then responds with a set of realistic values which are stored in a dictionary for subsequent use while generating tests. Practically, we deployed a local instance of GPT4All [19] with the model `wizardlm-13b-v1.2.Q4_0.gguf`, described on the GPT4All website as the "best overall larger model." For each parameter in the OAS of an API, the LLM is asked to provide at least 20 relevant values.

4.3 Mutation-Based Test Intensification

During test case generation, which is guided by curiosity and experience, the moment a new operation is successfully tested for the first time, we shift from exploration to exploitation (or intensification).

This phase involves multiple replays of the successful HTTP request after it has been modified by applying a catalog of mutation operators. Some of them are *nominal mutators*, which alter the initial request while still satisfying all the constraints in the OAS (thus generating further potentially valid requests), while others are *error mutators*, which alter the request in a way that violates some constraints from the OAS (thus generating potentially invalid requests). We believe that an invalid request, resembling a valid one, holds the potential to traverse unexplored branches in the API source code, thereby contributing to increased test coverage and hopefully leading to more effective test cases. To build the catalog, we started with mutation operators from literature: some of them keeping HTTP requests valid [20]; while others turning HTTP requests invalid [3] (according to the OAS). We then added additional mutation operators such that each parameter of each operation can be changed once to keep the overall request valid and once to turn it invalid. Currently, 10 mutation operators are available in DeepREST (4 nominal mutators and 6 error mutators), described in the following.

`AddParameter` An optional parameter is added to the original request. [*nominal*]

`RemoveParameter` An optional parameter is removed from the original request. [*nominal*]

`RefillValue` The value of a parameter is updated with a new value that satisfies the constraints in the OAS. [*nominal*]

`NumberBoundaries` In the case of a numeric parameter, its value is changed to be close to its boundaries. [*nominal*]

`AddInvalidParameter` An optional parameter with an invalid value is added to the original request. [*error*]

`NumberOutOfBoundaries` In the case of a numeric parameter, its value is changed to out of its boundaries. [*error*]

`ChangeHttpMethod` The HTTP method of the original request is changed. [*error*]

`MissingRequired` A mandatory parameter is removed from the original request. [*error*]

`WrongType` A parameter value is replaced with a new one of a different type. [*error*]

`ConstraintsViolation` A parameter value is changed so that it violates the constraints from the OAS. [*error*]

4.4 Oracles

DeepREST utilizes three standard oracles provided by the RestTestGen Framework [18]. In particular, it employs two types of oracles for nominal testing: a *Status Code Oracle* that identifies a bug in the API under test when it returns a `5XX` status code; and a *Schema Validation Oracle* that raises an issue when the API response schema is not valid according to the schema outlined in the OpenAPI specification. It also employs an oracle for error testing (i.e., when error mutation operators are applied to requests) based on the response status code. This oracle will report a failure in case the API processes a malformed requests without raising an error (i.e., the API replies with a `2XX` status code to a request not adhering to the OAS).

5 EVALUATION

In this section, we conduct a thorough empirical evaluation of DeepREST. Our primary focus is assessing the *test coverage* achieved by our approach and its *fault detection* capability. Test coverage, intended as both code coverage and operations successfully tested, is directly correlated with Challenges 1 and 2 presented in Section 3. Indeed, an optimal testing order of operations (Challenge 1) is likely to result in testing more operations successfully. Similarly, an appropriate and diverse input value selection (Challenge 2) is likely to result in testing more parts of the API implementation, increasing code coverage. Alongside successfully testing operations, to individuate faults a balance between exploration and exploitation (Challenge 3 of Section 3) is needed, testing again operations with different input values potentially inducing server-side errors. Since HTTP interactions are time-consuming, *testing efficiency*, in terms of the number of requests, is also investigated. To validate the aforementioned aspects, we engage in a comparative analysis with the current state-of-the-art testing tools for REST APIs.

5.1 Research Questions

To guide our empirical evaluation, we formulated the following two research questions.

RQ$_1$: What is the *effectiveness* of DeepREST in generating test cases for REST APIs? How does it compare with state-of-the-art approaches?

RQ$_2$: What is the *effeciency* of DeepREST in generating test cases for REST APIs? How does it compare with state-of-the-art approaches?

To answer *RQ$_1$*, we execute DeepREST and five state-of-the-art REST API testing tools on a benchmark set of API case studies. For each tool, targeting each API in the benchmark, we measure code coverage, count successfully tested operations (`2XX`), and server-side failures (`5XX`).

To answer *RQ$_2$*, for each experiment of the previous research question, we monitor code coverage and success/failure count over time to measure their progressive increase. The efficiency of a tool is deemed higher if it achieves high code coverage or success/failure count earlier in the testing process.

Table 1: Benchmark APIs.

API	Short Name	# Ops	LoC (JaCoCo)	LoC (IntelliJ)	Source
REST Countries	rest-countr	22	543	1,121	[7]
User Management	user-mgmt	22	632	1,284	[7]
Market	market	13	2,206	5,543	[7]
Project Tracking System	proj-track	59	1,298	3,613	[7]
Features Service	feat-serv	18	457	956	[7]
NCS	ncs	6	275	500	[7]
SCS	scs	11	295	586	[7]
Genome-Nexus	genome-nex	23	4,831	15,541	[7]
Person Controller	person-ctrl	12	179	522	[7]
Blog	blog	52	1,188	3,725	[21]
LanguageTool	lang-tool	2	45,487	83,708	[7]

5.2 Experiment Setup

Baseline Testing Tools. Our evaluation consists of a comparison of DeepREST with state-of-the-art REST API testing tools. We selected a total of five other tools. They are RestTestGen [3], Morest [4] and Restler [2] as representative of data dependencies-based approaches, ARAT-RL [7] as the only existing tool based on reinforcement learning (although using the non-deep algorithm Q-Learning), and Schemathesis [22] as a recently proposed tool whose performance seems competitive [7]. According to recent surveys [7, 20], ARAT-RL and RestTestGen appear to be the best-performing black-box tools.

API Case Studies. To establish a benchmark for our evaluation, we selected 11 API case studies. We included all the 10 APIs sourced from a recent study [7]. To enhance the realism of our evaluation, we supplemented this set with an additional API from GitHub (i.e., *Blog*) as representative of more complex APIs. In fact, it contains implicit operation dependencies that cannot be syntactically deduced from the OAS (we found such implicit dependencies by manually reverse engineering the API business logic).

This benchmark's size and complexity align with the literature on REST API testing. Moreover, we believe it is representative since the included APIs present the following characteristics: they have diverse sizes (number of operations and parameters); they come from diverse application domains; and they have diverse business logic complexities (to be extensively tested, they require API operation sequences of diverse lengths).

Table 1 reports the list of API case studies, accompanied by the number of operations defined in their respective OAS, the number of lines of code as reported by JaCoCo (which analyzes the bytecode in the compiled `.class` files) and number of source lines of code from the *Statistic*[2] plug-in for IntelliJ IDEA. We launched the plug-in on the "main" source code of each API, thus excluding the code in the `test` folder which is conventionally used to store test cases. The reported numbers refer to the *source code lines*, excluding comment lines and blank lines.

Metrics. We measured the following metrics throughout all testing sessions conducted with all tools and configurations.

Code Coverage We collected method, branch, and line coverage to measure the extent of API code executed by the test cases.

Operation Coverage This black-box coverage metric [23, 24] is meant to count the number of successfully tested API operations by tools with respect to the total number of operations defined in the API documentation.

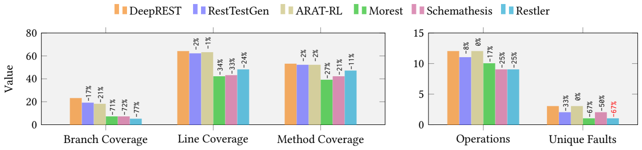

Figure 3: Effectiveness results (aggregate).

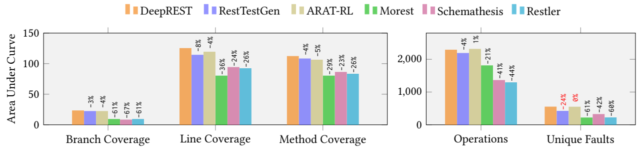

Figure 4: Efficiency results (aggregate).

Fault Detection As a measure of fault detection capacity, we counted the number of server-side failures (status code `5XX`) with unique error messages identified by tools. An error message is considered unique when it is sufficiently different from other messages, as previously defined by Kim et al. [7].

Area Under Curve As an efficiency metric, we captured the progress of code coverage, operation coverage, and fault detection while testing. This is measured as the area under the curve in a graph depicting test coverage and fault detection trends.

Experimental Procedure. Experiments have been run using Docker containers, with a separate container for each testing tool and for each case study. Each container is assigned a maximum of 8 cores and 16 GB of RAM. At the end of each experiment run, containers were stopped and rebooted with a fresh file system to avoid any side effects, either in tools or API case studies. Testing tools were executed with the same budget of API calls for each API in the benchmark. This ensures that all the tools have the same opportunity to explore and test the APIs, regardless of how long an API can non-deterministically take to respond. The budget has been computed for each API independently by empirically checking the amount of interactions that each API can serve in approximately one hour of testing. To control the impact of non-deterministic features of testing tools, all experiments have been repeated 10 times, reporting the average results. JaCoCo [25] was deployed to collect source code coverage, while Restats [24] was utilized for computing the operation coverage from HTTP logs. Metrics were recorded every 5 seconds.

5.3 Experiment Results

The results of our evaluation are reported in Figures 3, 4, 5 and 6. To test for statistical significance of observed differences, we apply the Wilcoxon signed rank test to compare the metrics values achieved by DeepREST with those by each tool in the comparison, computing the corresponding p-values. We assume a significance level of 95% ($\alpha = 0.05$), that is, we reject the null hypothesis when p-value < 0.05. Values of statistically significant differences are in black in the figures, while non-significant differences are in red.

RQ_1 (Effectiveness). The experimental results illustrating the effectiveness of DeepREST are presented in Figure 3 and Figure 5,

[2] https://plugins.jetbrains.com/plugin/4509-statistic

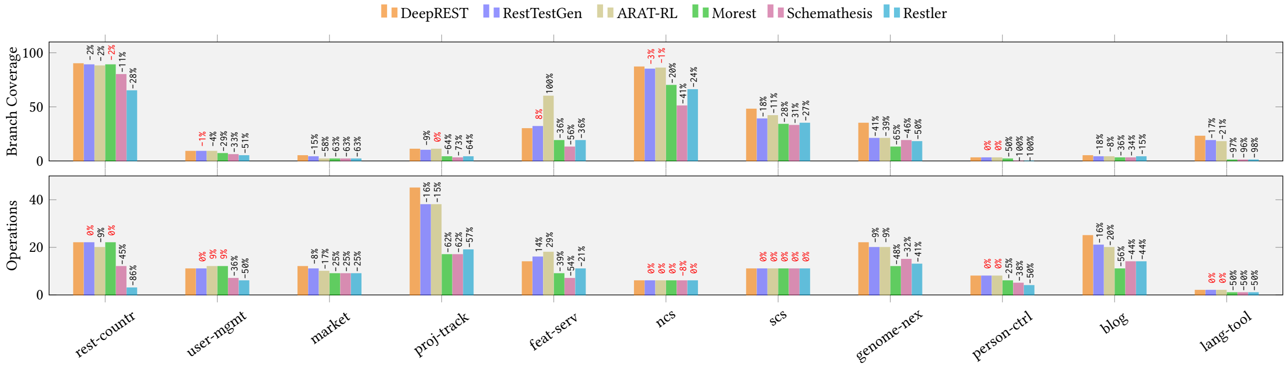

Figure 5: Effectiveness results (per API).

alongside comparable results from the tools in the comparison for reference. Figure 3 shows the overall results aggregated by tool, with different bar colors for different tools (DeepREST is the left-most bar in orange). On the left-hand side of the figure, we report the average branch, line and method coverage among all 11 APIs for each tool. On the right-hand side, we report the average count of successfully tested operations (2XX) and faults (5XX).

As we can see from the figure, DeepREST achieves the highest values for all the metrics among all the considered testing tools, and all the differences are statistically significant according to the Wilcoxon signed rank test (with the only exception of the faults for Restler). The most remarkable difference is for branch coverage that spans from 17% for RestTestGen to 77% for Restler.

Figure 5 represents the collected metrics results split by case study. For space reasons, we report the results for two metrics only (results for the other metrics are provided in the replication package [8]). The first bar graph represents branch coverage, while the second bar graph the successful operations count. This figure confirms the previously observed trend, with DeepREST outperforming all the other tools in almost all the case study APIs. The very few exceptions include the case of *Feature Service* API, for which ARAT-RL and RestTestGen achieve an higher score.

These results suggest that our approach achieved the highest effectiveness because it tested API states that other approaches could not reach. This is probably due to the fact that DeepREST learned how to assemble more effective sequences of operation calls that could not be assembled just by relying on (producer-consumer) dependencies documented in the API specification.

An interesting case to comment on is *Project Tracking System*, where DeepREST largely overcame other tools (16% to 62% higher coverage). The Operation Dependency Graph produced by RestTestGen for this API contains 62 operations and over 6000 data dependencies. On the one hand, this overwhelming large set of data dependencies probably contains many spurious dependencies, tricking testing tools into building unsuccessful sequences. On the other hand, there are so many dependencies that attempting all the resulting operation orders is quite inefficient. The advantage of deep reinforcement learning in this API is that whenever the correct order to test N operations is found, it is learned and reused to try and test a sequence of $N + 1$ operations without wasting the testing budget.

Specifically, in the *Project Tracking System* API, only after creating new `credentials` an `employee` can be created. Then, a `project` must be created. Only after successfully taking the API to this state the `POST␣/assignment` operation can be tested, resulting in a mandatory sequence of four operations. As a matter of fact, DeepREST gradually learned this operation ordering, progressively building longer and longer sequences, thus testing more operations on this API.

These results allow us to formulate the following answer to the first research question.

> ***Answer to RQ$_1$***: DeepREST is the most effective black-box testing tool for REST APIs, demonstrating higher effectiveness than state-of-the-art with respect to branch, line and method coverage, as well as successfully tested operations and revealed unique faults.

RQ$_2$ (Efficiency). The experimental results about the efficiency of DeepREST are presented in Figure 4 and Figure 6. Figure 4 shows the average results aggregated by tool, with different bar colors for different tools (DeepREST is the left-most bar in orange). On the left-hand side of the figure, we report the average Area Under Curve (AUC) for branch, line, and method coverage among all 11 APIs for each tool. On the right-hand side, we report the AUC for the successfully tested operations (2XX) and the faults (5XX) count.

As we can see from the figure, DeepREST is more efficient in achieving high values for all the metrics than all the other testing tools, with the only exception of ARAT-RL for the successful operations count. Moreover, almost all the differences are statistically significant according to the Wilcoxon signed rank test (with the only exception of unique faults by RestTestGen and ARAT-RL). The most remarkable difference is for the AUC of the successfully tested operations count that spans from 4% of RestTestGen to 44% of Restler, and line coverage that spans between 4% (ARAT-RL) to 36% (Morest).

Figure 6 shows the results for the AUC of the same metrics, split by case study. For space reasons, we report the results for two metrics only (results for the other metrics are provided in the replication package [8]). The first bar graph represents branch coverage, while the second bar graph the successful operations count. DeepREST appear consistently superior state-of-the-art tools in most of the cases, with few exceptions (already observed for effectiveness); they are ARAT-RL and RestTestGen on the branch coverage of *Feature Service*.

Based on these results, we can formulate the following answer to the second research question.

> ***Answer to RQ_2***: DeepREST demonstrates higher efficiency than state-of-the-art testing tools concerning branch, line and method coverage. Regarding successfully tested operations and revealed unique faults, DeepREST is superior to most state-of-the-art tools except ARAT-RL.

5.4 Threats to Validity

Threats to *internal validity*, impacting empirical results, are due to the metrics adopted to answer the research questions and the configuration of the tools in the comparison. To mitigate these threats, we adopted standard metrics from structural testing (code coverage) and specific metrics for REST API testing (operation coverage [23] and fault detection[7]). Furthermore, the latest versions of the considered testing tools have been used in the comparison, with tools configured as indicated in the corresponding papers. We addressed potential threats due to randomness issues by running each tool 10 times and computing the average results. Moreover, we applied a statistical test (i.e., Wilcoxon signed rank test) that is non-parametric, thus it does not assume experimental data to be normally distributed.

Threats to *external validity*, impacting the generalization of our findings, are due to the case studies selected for the tools comparison and their limited number. We mitigated these threats by considering the dataset of REST APIs adopted in previous studies [7], in addition to one new API.

6 RELATED WORK

Among automated REST API testing tools, only EvoMaster [26] adopts a white-box approach. Specifically, test case generation is guided by evolutionary algorithms, whose fitness function is defined in terms of API code coverage and HTTP interactions status code. EvoMaster also provides a black-box version of the tool that only resorts to the API documentation to guide generation.

Literature about black-box REST API testing is broader, comprising various tools implementing different testing strategies. We already mentioned approaches, such as RESTler [2] and RestTestGen [3], that use data dependencies to prioritize operations testing. RESTler generates sequences of HTTP interactions by exploiting producer-consumer dependencies contained in the OAS, targeting internal server failures. Instead, RestTestGen exploits the Operation Dependency Graph, embedding data dependencies between operations, to craft meaningful test cases in nominal and error scenarios. As already pointed out, such approaches might be biased by spurious data dependencies or miss implicit operation dependencies that are instead exploited by DeepREST.

From a general viewpoint, QuickREST [27] performs property-based testing of REST APIs, generating test cases with the aim of verifying whether an API complies with some properties documented in its OAS. Similarly, Schemathesis [22] detects faults by checking response compliance in OpenAPI or GraphQL APIs via property-based testing. Morest [4] exploits a Property Graph, dynamically updated during testing, to model the behavior of the API under test. Knowledge from the graph is used to craft meaningful test sequences. RESTest [28] is a tool that provides inter-parameter dependencies testing, producing nominal and faulty test cases. RestCT [29] leverages combinatorial testing to generate test cases for REST APIs based on the OAS. Dredd [30] is a tool testing REST APIs by comparing actual responses with expected ones, checking their status code, header, and body. Tcases [31] is a model-based tool, leveraging the OAS to systematically build an input space model. Subsequently, it generates test cases covering valid input dimensions and checking response status codes for validation.

REST API fuzzers [32–36] are black-box tools that generate new test cases starting from previously recorded HTTP traffic: they fuzz and replay new HTTP requests in order to find faults. Some of them [33–35] also exploit OASs.

Reinforcement learning has been recently adopted in software testing, focusing primarily on web and mobile applications. In particular, Zheng et al. [14] and Pan et al. [15] propose automatic testing approaches based on curiosity-driven reinforcement learning for web clients and Android apps, respectively. Vuong and Takada [37] and Koroglu et al. [38] also apply reinforcement learning to automated testing of Android apps, the latter adopting an exploration method based on Q-Learning. Adamo et al. [39] present a reinforcement learning-based technique specifically designed for Android GUI testing, while Koroglu and Sen [40] present a reinforcement learning-based method for generating functional tests from UI test scenarios for Android apps. Mariani et al. [41] proposed AutoBlackTest, an automatic black-box testing approach for interactive applications. Deep reinforcement learning has been used by Romdhana et al. to perform black-box functional testing [42]; security testing [43] of Android apps, and focus testing of C code [44].

The closest work is ARAT-RL [7], which exploits reinforcement learning to test REST APIs. Utilizing Q-Learning, ARAT-RL determines the priority of operations to test initially by considering the frequency of parameters in the API specification (that can be seen as a sort of data dependencies-based initialization). It subsequently refines this prioritization based on the HTTP interactions with the API. Nevertheless, such refinement is limited by the learning strategy adopted by ARAT-RL: agents always receive a high penalty when successfully tested operations are considered again. This promotes exploration of not yet tested operations only, discouraging agents from crafting complex sequences of operations. Indeed, to spot implicit dependencies, considering already successfully tested operations is crucial since they may trigger operations that otherwise are unlikely to be exercised.

In DeepREST, instead, deep reinforcement learning starts with an empty experience, avoiding any possible bias from data dependencies. Moreover, DeepREST explicitly models (an approximation of) the internal state of the REST API under test, and agents are encouraged to test operations multiple times in different API states. This may yield corner case preconditions triggering hard-to-test operations, corresponding to implicit dependencies that can be spotted by crafting complex sequences of operation invocations

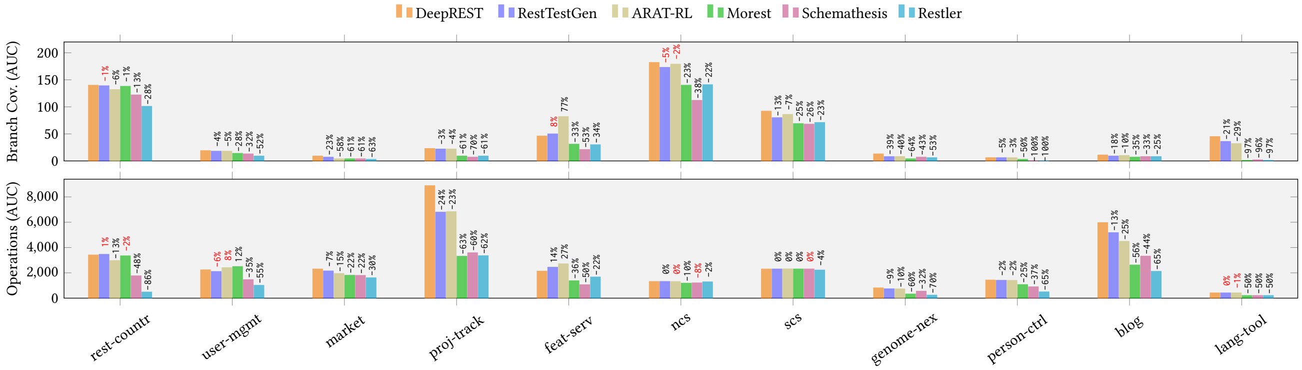

Figure 6: Efficiency results (per API).

only. This is possible due to the DeepREST API state representation encoding the history of past successes, which is used as a guide for operation sequencing. This complex state representation can not be modeled by Q-Learning and requires a *deep* reinforcement learning approach.

7 CONCLUSION

Black-box REST API testing tools ground their test case generation strategies on the information contained in the OAS of the API under test. This poses limitations on those APIs that contain implicit operation dependencies and value conditions. Such constraints cannot be retrieved from the OAS, making testing tools blind with respect to potentially crucial parts of the API business logic.

In this paper, we showed how such hidden API constraints can be learned from API interactions attempted at testing time, even in a black-box setting. We indeed proposed the first REST API testing approach based on deep reinforcement learning, having the twofold objective of computing an effective ordering of API operations to test, encompassing (explicit and implicit) operation dependencies, and selecting accurate input data for operation parameters. Operations ordering inference is guided by *exploration* of the API under test, while input data generation leverages *experience* gained on successful API interactions. Finally, DeepREST *intensifies* testing by mutating successful API interactions in order to achieve higher test coverage and collaterally increasing experience.

Empirical evidence showed that the proposed approach results in boosting the performance of REST API testing. Indeed, DeepREST is shown to overcome state-of-the-art approaches, both in terms of effectiveness and efficiency.

In future work, we aim to enhance experience-driven input generation by enabling the learning of inter-parameter dependencies. Additionally, we seek to explore the influence of input generation techniques on the overall effectiveness of the generated test cases. In particular, we plan to assess the individual contributions of the adopted sources of input values, including the LLM-generated dictionary.

DATA AVAILABILITY

All the material needed to replicate our experiments is available on Zenodo [8]. The latest version of DeepREST can be found in the RestTestGen GitHub repository [9], where it is implemented as a custom testing strategy.

ACKNOWLEDGEMENTS

This research was partially supported by the European Union’s Horizon Europe research and innovation programme, under grant 101070238; by European Union under NextGenerationEU with the *SMARTITUDE* research project, which has been funded by MUR under the PRIN 2022 program (Code: 202233YFCJ); by the Italian Ministry of University and Research, under the PNRR programme for the Interconnected Nord-Est Innovation Ecosystem (iNEST); by the Italian Ministry of University and Research, under agreement 40-G-14702-3 for the PON programme for Research and Innovation (Action IV.6). This publication reflects the views only of the authors, and the sponsoring agencies cannot be held responsible for such views and any use which may be made of the information contained therein.

REFERENCES

[1] A. Golmohammadi, M. Zhang, and A. Arcuri, “Testing restful apis: A survey,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, nov 2023.

[2] V. Atlidakis, P. Godefroid, and M. Polishchuk, “RESTler: Stateful REST API fuzzing,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE ’19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 748–758.

[3] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, “Automated black-box testing of nominal and error scenarios in RESTful APIs,” *Software Testing, Verification and Reliability*, Jan. 2022.

[4] 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*. New York, NY, USA: ACM, 2022, pp. 1406–1417.

[5] V. Atlidakis, P. Godefroid, and M. Polishchuk, “Checking security properties of cloud service REST APIs,” in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020.* IEEE, 2020, pp. 387–397.

[6] A. Martin-Lopez, S. Segura, C. Muller, and A. Ruiz-Cortes, “Specification and automated analysis of inter-parameter dependencies in web APIs,” *IEEE Transactions on Services Computing*, pp. 1–1, 2021.

[7] M. Kim, S. Sinha, and A. Orso, “Adaptive REST API testing with reinforcement learning,” in *IEEE/ACM International Conference on Automated Software Engineering*, 2023.

[8] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato. (2024) Replication package for the paper “DeepREST: Automated test case generation for REST APIs exploiting deep reinforcement learning”. [Online]. Available: https://zenodo.org/records/11525389

[9] SeUniVr. (2024) RestTestGen. [Online]. Available: https://github.com/SeUniVr/RestTestGen

[10] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000, vol. 7.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[12] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, may 1992.

[13] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[14] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 423–435.

[15] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2020, pp. 153–164.

[16] OpenAI, "Proximal policy optimization," 2023, https://openai.com/research/openai-baselines-ppo.

[17] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[18] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "Resttestgen: An extensible framework for automated black-box testing of restful apis," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 504–508.

[19] nomic-ai. (2024) GPT4All. [Online]. Available: https://gpt4all.io

[20] 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*, 2023, pp. 1232–1243.

[21] osopromadze. (2024) Blog REST API. [Online]. Available: https://github.com/osopromadze/Spring-Boot-Blog-REST-API

[22] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web api schemas," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA: ACM, 2022, pp. 345–346.

[23] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Test coverage criteria for RESTful web APIs," in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 15–21.

[24] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "Restats: A test coverage tool for RESTful APIs," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 594–598.

[25] Jacoco. (2023) JaCoCo Java Code Coverage Library. [Online]. Available: https://github.com/jacoco/jacoco

[26] A. Arcuri, "RESTful API automated test case generation with Evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, p. 3, 2019.

[27] S. Karlsson, A. Causevic, and D. Sundmark, "QuickREST: Property-based test generation of OpenAPI-described RESTful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 131–141.

[28] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-box constraint-based testing of RESTful web APIs," in *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings*, ser. Lecture Notes in Computer Science, E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, and H. Motahari, Eds., vol. 12571. Springer, 2020, pp. 459–475.

[29] H. Wu, L. Xu, X. Niu, and C. Nie, "Combinatorial testing of restful apis," in *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: ACM, 2022, pp. 426–437.

[30] apiaryio. (2023) Dredd. [Online]. Available: https://github.com/apiaryio/dredd

[31] Cornutum. (2023) Tcases. [Online]. Available: https://github.com/Cornutum/tcases

[32] KissPeter. (2022) API Fuzzer. [Online]. Available: https://github.com/KissPeter/APIFuzzer

[33] Yelp. (2022) Fuzz-Lightyear. [Online]. Available: https://github.com/Yelp/fuzz-lightyear

[34] namuan. (2022) Fuzzy-Swagger. [Online]. Available: https://github.com/namuan/fuzzy-swagger

[35] Lothiraldan. (2022) Swagger-Fuzzer. [Online]. Available: https://github.com/Lothiraldan/swagger-fuzzer

[36] Teebytes. (2022) TnT-Fuzzer. [Online]. Available: https://github.com/Teebytes/TnT-Fuzzer

[37] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. New York, NY, USA: ACM, 2018, pp. 31–37.

[38] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "QBE: Qlearning-based exploration of android applications," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 105–115.

[39] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. New York, NY, USA: ACM, 2018, pp. 2–8.

[40] Y. Koroglu and A. Sen, "Functional test generation from ui test scenarios using reinforcement learning for android applications," *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1752, 2021.

[41] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic black-box testing of interactive applications," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 81–90.

[42] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–29, 2022.

[43] ——, "Assessing the security of inter-app communications in android through reinforcement learning," *Computers & Security*, vol. 131, p. 103311, 2023.

[44] A. Romdhana, M. Ceccato, A. Merlo, and P. Tonella, "Ifrit: Focused testing through deep reinforcement learning," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 24–34.