Adaptive REST API Testing with Reinforcement Learning

Myeongsoo Kim
Georgia Institute of Technology
Atlanta, Georgia, USA
mkim754@gatech.edu

Saurabh Sinha
IBM Research
Yorktown Heights, New York, USA
sinhas@us.ibm.com

Alessandro Orso
Georgia Institute of Technology
Atlanta, Georgia, USA
orso@cc.gatech.edu

***Abstract*—Modern web services increasingly rely on REST APIs. Effectively testing these APIs is challenging due to the vast search space to be explored, which involves selecting API operations for sequence creation, choosing parameters for each operation from a potentially large set of parameters, and sampling values from the virtually infinite parameter input space. Current testing tools lack efficient exploration mechanisms, treating all operations and parameters equally (i.e., not considering their importance or complexity) and lacking prioritization strategies. Furthermore, these tools struggle when response schemas are absent in the specification or exhibit variants. To address these limitations, we present an adaptive REST API testing technique that incorporates reinforcement learning to prioritize operations and parameters during exploration. Our approach dynamically analyzes request and response data to inform dependent parameters and adopts a sampling-based strategy for efficient processing of dynamic API feedback. We evaluated our technique on ten RESTful services, comparing it against state-of-the-art REST testing tools with respect to code coverage achieved, requests generated, operations covered, and service failures triggered. Additionally, we performed an ablation study on prioritization, dynamic feedback analysis, and sampling to assess their individual effects. Our findings demonstrate that our approach outperforms existing REST API testing tools in terms of effectiveness, efficiency, and fault-finding ability.**

***Index Terms*—Reinforcement Learning for Testing, Automated REST API Testing**

I. Introduction

The increasing adoption of modern web services has led to a growing reliance on REpresentational State Transfer (REST) APIs for communication and data exchange [1], [2]. REST APIs adhere to a set of architectural principles that enable scalable, flexible, and efficient interactions between various software components through the use of standard HTTP methods and a stateless client-server model [3]. To facilitate their discovery and use by clients, REST APIs are often documented using various specification languages [4]–[7] that let developers describe the APIs in a structured format and provide essential information, such as the available endpoints, input parameters and their schemas, response schemas, and so on. Platforms such as APIs Guru [8] and Rapid [9] host thousands of RESTful API documents, emphasizing the significance of formal API specifications in industry.

Standardized documentation formats, such as the OpenAPI specification [4], not only facilitate the development of REST APIs and their use by clients, but also provide a foundation for the development of automated testing techniques for such APIs, and numerous such techniques and tools have emerged in recent years (e.g., [10]–[17]). In spite of this, testing REST APIs continues to be a challenging task, with high code coverage remaining an elusive goal for automated tools [18].

Testing REST APIs can be challenging because of the large search space to be explored, due to the large number of operations, potential execution orders, inter-parameter dependencies, and associated input parameter value constraints [18], [19]. Current techniques often struggle when exploring this space due to lack of effective exploration strategies for operations and their parameters. In fact, existing testing tools tend to treat all operations and parameters equally, disregarding their relative importance or complexity, which leads to suboptimal testing strategies and insufficient coverage of crucial operation and parameter combinations. Moreover, these tools rely on discovering producer-consumer relationships between response schemas and request parameters; this approach works well when the parameter and response schemas are described in detail in the specification, but falls short in the common case in which the schemas are incomplete or imprecise.

To address these limitations of the state of the art, we present adaptive REST API testing with reinforcement learning (ARAT-RL), an advanced black-box testing approach. Our technique incorporates several innovative features, such as leveraging reinforcement learning to prioritize operations and parameters for exploration, dynamically constructing key-value pairs from both response and request data, analyzing these pairs to inform dependent operations and parameters, and utilizing a sampling-based strategy for efficient processing of dynamic API feedback. The primary objectives of our approach are to increase code coverage and improve fault-detection capability.

The core novelty in ARAT-RL is an adaptive testing strategy driven by a reinforcement-learning-based prioritization algorithm for exploring the space of operations and parameters. The algorithm initially determines the importance of an operation based on the parameters it uses and the frequency of their use across operations. This targeted exploration enables efficient coverage of critical operations and parameters, thereby improving code coverage. The technique employs reinforcement learning to adjust the priority weights associated with operations and parameters based on feedback, by

DOI 10.1109/ASE56229.2023.00218

decreasing importance for successful responses and increasing it for failed responses. The technique also assigns weights to parameter-value mappings based on various sources of input values (e.g., random, specified values, response values, request values, and default values). The adaptive nature of ARAT-RL gives precedence to yet-to-be-explored or previously error-prone operations, paired with suitable value mappings, which improves the overall efficiency of API exploration.

Another innovative feature of ARAT-RL is dynamic construction of key-value pairs. In contrast to existing approaches that rely heavily on resource schemas provided in the specification, our technique dynamically constructs key-value pairs by analyzing POST operations (i.e., the HTTP methods that create resources) and examining both response and request data. For instance, suppose that an operation takes book title and price as request parameters and, as response, produces a success status code along with a string message (e.g., “Successfully created”). Our technique leverages this information to create key-value pairs for book title and price, upon receiving a successful response, even if such data is not present in the response. In other words, the technique takes into account the input parameters used in the request, as they correspond to the created resource. Moreover, if the service returns incomplete resources (i.e., only a subset of the data items for a given type of resource), our technique still creates key-value pairs for the information available. This dynamic approach enables ARAT-RL to identify resources from the API responses and requests, as well as discover hidden dependencies that are not evident from the specification alone.

Finally, ARAT-RL employs a simple yet effective sampling-based approach that allows it to process dynamic API feedback efficiently and adapt its exploration based on the gathered information. By randomly sampling key-value pairs from responses, our technique reduces the overhead of processing every response for each pair, resulting in more efficient testing.

To evaluate ARAT-RL, we conducted a set of empirical studies using 10 RESTful services and compared its performance against three state-of-the-art REST API testing tools: RESTler [12], EvoMaster [10], and Morest [20]. We assessed the effectiveness of ARAT-RL in terms of coverage achieved and service failures triggered, and its efficiency in terms of valid and fault-inducing requests generated and operations covered within a given time budget. Our results show that ARAT-RL outperforms the competing tools for all the metrics considered—it achieved the highest method, branch, and line coverage rates, along with better fault-detection ability. Specifically, ARAT-RL covered 119%, 60%, and 52% more branches, lines, and methods than RESTler; 37%, 21%, and 14% more branches, lines, and methods than EvoMaster; and 24%, 12%, and 10% more branches, lines, and methods than Morest. ARAT-RL also uncovered 9.3x, 2.5x, and 2.4x more bugs than RESTler, EvoMaster, and Morest, respectively. In terms of efficiency, ARAT-RL generated 52%, 41%, and 1,222% more valid and fault-inducing requests and covered 15%, 24%, and 283% more operations than Morest, EvoMaster, and RESTler, respectively. We also conducted an ablation study to assess the individual effects of prioritization, dynamic feedback analysis, and sampling on the overall effectiveness of ARAT-RL. Our results indicate that reinforcement-learning-based prioritization contributes the most to ARAT-RL’s effectiveness, followed by dynamic feedback analysis and sampling.

```
/products/{productName}/configurations/{configurationName}/features/{featureName}:
  post:
      operationId: addFeatureToConfiguration
      produces:
        - application/json
      parameters:
        - name: productName
          in: path
          required: true
          type: string
        - name: configurationName
          in: path
          required: true
          type: string
        - name: featureName
          in: path
          required: true
          type: string
      responses:
        default:
          description: successful operation
/products/{productName}/configurations/{configurationName}/features:
    get:
      operationId: getConfigurationActivedFeatures
      produces:
        - application/json
      parameters:
        - name: productName
          in: path
          required: true
          type: string
        - name: configurationName
          in: path
          required: true
          type: string
      responses:
        '200':
          description: successful operation
          schema:
            type: array
            items:
              type: string
```

Fig. 1. A Part of Features Service’s OpenAPI Specification.

The main contributions of this work are:

- A novel approach for adaptive REST API testing that incorporates (1) reinforcement learning to prioritize exploration of operations and parameters, (2) dynamic analysis of request and response data to identify dependent parameters, and (3) a sampling strategy for efficient processing of dynamic API feedback.
- Empirical results demonstrating that ARAT-RL outperforms state-of-the-art REST API testing tools by generating more valid and fault-inducing requests, covering more operations, achieving higher code coverage, and triggering more service failures.
- An artifact [21] containing the tool, the benchmark services, and the empirical results.

The rest of this paper is organized as follows. Section II presents background information and a motivating example to illustrate challenges in REST API testing. Section III describes our approach. Section IV presents our empirical evaluation and results. Section V discusses related work and, finally, Section VI presents our conclusions and potential directions for future work.

II. Background and Motivating Example

We provide a brief introduction to REST APIs, OpenAPI specifications, and reinforcement learning; then, we illustrate the novel features of our approach using a running example.

A. REST APIs

REST APIs are web APIs that adhere to the RESTful architectural style [3]. REST APIs facilitate communication between clients and servers by exchanging data through standardized protocols, such as HTTP [22]. Key principles of REST include statelessness, cacheability, and a uniform interface, which simplify client-server interactions and promote loose coupling [23]. Clients communicate with web services by sending HTTP requests. These requests access and/or manipulate resources managed by the service, where a resource represents data that a client may want to create, delete, update, or access. Requests are sent to an API endpoint, identified by a resource path and an HTTP method specifying the action to be performed on the resource. The most commonly used methods are POST, GET, PUT, and DELETE, for creating, reading, updating, and deleting a resource, respectively. The combination of an endpoint and an HTTP method is called an operation. Besides specifying an operation, a request can also optionally include HTTP headers containing metadata and a body with the request's payload. Upon receiving and processing a request, the web service returns a response containing headers, possibly a body, and an HTTP status code—a three-digit number indicating the request's outcome. Specifically, 2xx status codes denote successful responses, 4xx codes indicate client errors, and status code 500 suggests server error.

B. OpenAPI Specification

The OpenAPI Specification (OAS) [4] is a widely adopted API description format for RESTful APIs, providing a standardized and human-readable way to describe the structure, functionality, and expected behavior of an API. Figure 1 illustrates an example OAS file describing a part of the Features Service API. This example shows two API operations. The first operation, a POST request, is designed to add a feature name to a product's configuration. It requires three parameters: product name, configuration name, and feature name, all of which are specified in the path. Upon successful execution, the API responds with a JSON object, signaling that the feature has been added to the configuration. The second operation, a GET request, retrieves the active features of a product's configuration. Similar to the first operation, it requires the product name and configuration name as path parameters. The API responds with a 200 status code and an array of strings representing the active features in the specified configuration.

C. Reinforcement Learning and Q-Learning

Reinforcement learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment [24]. The agent selects actions in various situations (states), observes the consequences (rewards), and learns to choose the best actions to maximize the cumulative reward over time. The learning process in RL is trial-and-error based, meaning the agent discovers the best actions by trying out different options and refining its strategy based on the observed rewards. The agent must also decide between exploring new actions to gather more knowledge or exploiting known actions that offer the best reward based on its current understanding. The balance between exploration and exploitation is often governed by parameters, such as ϵ in the ϵ-greedy strategy [24].

Q-learning is a widely used model-free reinforcement learning algorithm that estimates the optimal action-value function, $Q(s, a)$ [25]. The Q-function represents the expected cumulative reward the agent can obtain by taking action a in state s and then following the optimal policy. Q-learning uses a table to store Q-values and updates them iteratively based on the agent's experiences. In the learning process, the agent takes actions, receives rewards, and updates the Q-values using the Q-learning update rule, derived from the Bellman equation [24]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \quad (1)$$

where α is the learning rate, γ is the discount factor, s' is the new state after taking action a, and r is the immediate reward received. The agent updates the Q-values to converge to their optimal values, which represent the expected long-term reward of taking each action in each state.

D. Motivating Example

Next, we illustrate the salient features of ARAT-RL using the Feature-Service specification (Figure 1) as an example.

RL-based adaptive exploration: For the example in Figure 1, to perform the operation `addFeatureToConfiguration`, we must first create a product using a separate operation and establish a configuration for it using another operation. The sequence of operations should, therefore, be: create product, create configuration, and create feature name for the product with the specified configuration name. This example emphasizes the importance of determining the operation sequence. Our technique initially assigns weights to operations and parameters based on their usage frequency in the specification. In this case, `productName` is the most frequently used parameter across all operations; therefore, our technique assigns higher weights to operations involving `productName`. Specifically, the operation for creating a product gets the highest priority.

Moreover, once an operation is executed, its priority must be adjusted so that it is not explored repeatedly, creating new product instances unnecessarily. After processing a prioritized operation, our technique employs RL to adjust the weights in response to the API response received. If a successful response is obtained, negative rewards are assigned to the processed parameters, as our objective is to explore other uncovered operations. This method naturally leads to the selection of the next priority operation and parameter, facilitating efficient adjustments to the call sequence.

Inter-parameter dependencies [19] can increase the complexity of the testing process, as some parameters might have mutual exclusivity or other constraints associated with them (e.g., only one of the parameters can be specified). RL-based exploration guided by feedback received can also help with dealing with this complexity.

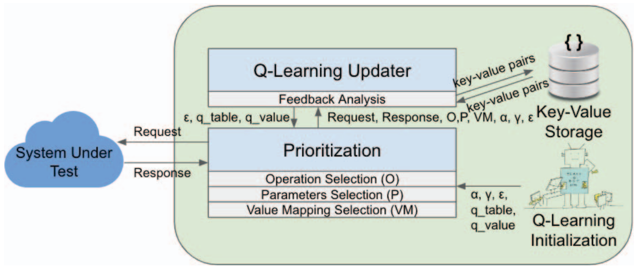

Fig. 2. Overview of our approach.

Dynamic construction of key-value pairs: Existing REST API testing strategies (e.g., [11], [12], [20]) emphasize the importance of identifying producer-consumer relationships between response schemas and request parameters. However, current tools face limitations when operations produce unstructured output (e.g., plain text) or have incomplete response schemas in their specifications. For instance, the `addFeatureToConfiguration` operation lacks structured response data (e.g., JSON format). Despite this, our approach processes and generates key-value data {`productName: <value>, configurationName: <value>, featureName: <value>`} from the request data, as the POST HTTP method indicates that a resource is created using the provided inputs.

By analyzing and storing key-value pairs identified from request and response data, our dynamic key-value pair construction method proves especially beneficial in cases of responses with plain-text descriptions or incomplete response schemas. The technique can effectively uncover hidden dependencies not evident from the specification.

Sampling for efficient dynamic key-value pair construction: API response data can sometimes be quite large and processing every response for each key-value pair can be computationally expensive. To address this issue, we have incorporated a samplingd strategy into our dynamic key-value pair construction method. This strategy efficiently processes the dynamic API feedback and adapts its exploration based on the gathered information while minimizing the overhead of processing every response.

III. Our Approach

In this section, we introduce our Q-Learning-based REST API testing approach, which intelligently prioritizes and selects operations, parameters, and value-mapping sources while dynamically constructing key-value pairs. Figure 2 provides a high-level overview of our approach. Initially, the Q-Learning Initialization module sets up the necessary variables and tables for the Q-learning process. Q-Learning Updater subsequently receives these variables and tables and passes them to the Prioritization module, which is responsible for selecting operations, parameters, and value-mapping sources.

ARAT-RL then sends a request to the System Under Test (SUT) and receives a response. It also supplies the request, response, selected operation, parameters, mapped value source, and the Q-Learning parameters (α, γ, and ϵ) to Q-Learning Updater. The feedback is analyzed with the request and response, storing key-value pairs extracted from them for future use. The Updater component then adjusts the Q-values based on the outcomes, enabling the approach to adapt its decision-making process over time. ARAT-RL iterates through this procedure until the specified time limit is reached. In the rest of this section, we present the details of the algorithm.

Algorithm 1 Q-Learning Table Initialization

```
 1: procedure INITIALIZEQLEARNING(operations)
 2:     Set learning rate ($\alpha$) to 0.1
 3:     Set discount factor ($\gamma$) to 0.99
 4:     Set exploration rate ($\epsilon$) to 0.1
 5:     Initialize empty dictionary q_table
 6:     Initialize empty dictionary q_value
 7:     for operation in operations do
 8:         operation_id ← operation['operationId']
 9:         q_value[operation_id] ← new dictionary
10:         for source in [S1, S2, S3, S4, S5] do
11:             q_value[operation_id][source] ← 0
12:         end for
13:         for parameter in operation['parameters'] do
14:             param_name ← parameter['name']
15:             if key in q_table then
16:                 q_table[param_name] = q_table[param_name] + 1
17:             else
18:                 q_table[param_name] = 1
19:             end if
20:         end for
21:         for response_data in operation_data.get('responses') do
22:             for key in response_data.keys() do
23:                 if key in q_table then
24:                     q_table[key] = q_table[key] + 1
25:                 end if
26:             end for
27:         end for
28:     end for
29:     return $\alpha$, $\gamma$, $\epsilon$, q_table, q_value
30: end procedure
```

A. *Q-Learning Table Initialization*

The Q-Learning Table Initialization component, shown in Algorithm 1, is responsible for setting up the initial Q-table and Q-value data structures that guide the decision-making process throughout a testing session. Crucially, this process happens without making any API calls.

The algorithm begins by setting the learning rate α to 0.1, the discount factor γ to 0.99, and the exploration rate ϵ to 0.1 (lines 2–4). These parameters control the learning and exploration process of the Q-Learning algorithm; the chosen values are the ones that are commonly recommended and used (e.g., [26]–[28]). The algorithm then initializes the Q-table and the Q-value with empty dictionaries (lines 5–6).

The algorithm iterates through each operation in the API (lines 7–24). For each operation, it extracts the operation's unique identifier (operation_id) and creates a new entry in the Q-value dictionary for the operation (lines 8–9). Next, it initializes the Q-value for each value-mapping source (S1–S5) to zero (lines 10–12).

The algorithm proceeds to iterate through each parameter in the operation (lines 13–20). It extracts the parameter's name (param_name) and, if param_name already exists in the Q-table, increments the corresponding entry by one; otherwise, it initializes the entry to one. This step builds the Q-table with counts of occurrences of each parameter.

Algorithm 2 Q-Learning-based Prioritization

```
 1: procedure SELECTOPERATION(operations, q_table)
 2:     Initialize max_avg_q_value ← $-\infty$
 3:     Initialize best_operation ← None
 4:     for operation in operations do
 5:         operation_id ← operation['operationId']
 6:         Initialize sum_q_value ← 0
 7:         Initialize num_params ← len(operation['parameters'])
 8:         for parameter in operation['parameters'] do
 9:             param_name ← parameter['name']
10:             sum_q_value ← sum_q_value + q_table[param_name]
11:         end for
12:         avg_q_value ← sum_q_value / num_params
13:         if avg_q_value > max_avg_q_value then
14:             max_avg_q_value ← avg_q_value
15:             best_operation ← operation
16:         end if
17:     end for
18:     return best_operation
19: end procedure
 1: procedure SELECTPARAMETERS(operation, $\epsilon$)
 2:     Set n randomly ($0 \leq n \leq$ length of operation['parameters'])
 3:     Initialize empty list selected_parameters
 4:     if random.random() > $\epsilon$ then
 5:         Sort operation['parameters'] by Q-values in descending order
 6:         for i ← 0 to n − 1 do
 7:             Append operation['parameters'][i] to selected_parameters
 8:         end for
 9:     else
10:         for param in random.sample(operation['parameters'], n) do
11:             Append param to selected_parameters
12:         end for
13:     end if
14:     return selected_parameters
15: end procedure
 1: procedure SELECTVALUEMAPPINGSOURCE(operation, $\epsilon$)
    Source1: Example values in specification
    Source2: Random value generated based on parameter's type and format
    Source3: Dynamically constructed key-value pairs from request
    Source4: Dynamically constructed key-value pairs from response
    Source5: Default values (string: string, number: 1.1, integer: 1, array: [], object: {})
 2:     operation_id ← operation['operationId']
 3:     sources ← [S1, S2, S3, S4, S5]
 4:     if random.random() > $\epsilon$ then
 5:         max_q_value ← $-\infty$
 6:         max_q_index ← −1
 7:         for s in sources do
 8:             if q_value[operation_id][s] > max_q_value then
 9:                 max_q_value ← q_value[operation_id][s]
10:                 max_q_index ← s
11:             end if
12:         end for
13:         return max_q_index
14:     else
15:         return random.randint(1, 5)
16:     end if
17: end procedure
```

Algorithm 3 Q-Learning-based API Testing

```
 1: procedure QLEARNINGUPDATER(response, q_table, q_value, selected_op, selected_params, $\alpha$, $\gamma$)
 2:     operation_id ← selected_op['operation_id']
 3:     if response.status_code is 2xx (successful) then
 4:         Extract key-value pairs from request and response
 5:         reward ← −1
 6:         Update q_value negatively
 7:     else if response.status_code is 4xx or 500 (unsuccessful) then
 8:         reward ← 1
 9:         Update q_value positively
10:     end if
11:     for each param in selected_params do
12:         for each param_name, param_value in param.items() do
13:             old_q_value ← q_table[operation_id][param_name]
14:             max_q_value_next_state ← max(q_table[operation_id].values())
15:             q_table[operation_id][param_name] ← old_q_value + $\alpha$ * (reward + $\gamma$ * (max_q_value_next_state - old_q_value))
16:         end for
17:     end for
18:     return q_table, q_value
19: end procedure
 1: procedure MAIN(API specification)
 2:     Initialize $\epsilon_{max} \leftarrow 1$
 3:     Initialize $\epsilon_{adapt} \leftarrow 1.1$
 4:     Initialize time_limit ← desired time limit in seconds
 5:     operations ← Load API specification
 6:     $\alpha$, $\gamma$, $\epsilon$, q_table, q_value ← INITIALIZEQLEARNING(operations)
 7:     while time_limit not reached do
 8:         operation ← SELECTOPERATION(operations, q_table)
 9:         parameters ← SELECTPARAMETERS(operation, $\epsilon$)
10:         source ← SELECTVALUEMAPPINGSOURCE(operation, $\epsilon$)
11:         response ← Execute API request with operation, parameters, and source
12:         q_table, q_value ← QLEARNINGUPDATER(response, q_table, q_value, operation, parameters, $\alpha$, $\gamma$)
13:         $\epsilon \leftarrow \min(\epsilon_{max}, \epsilon_{adapt} * \epsilon)$
14:     end while
15: end procedure
```

Next, the algorithm iterates through the response data of each operation (lines 21–27). It extracts keys from the response and checks, for each key, whether it is present in the Q-table for that operation (line 18). If a key is present, the algorithm increments the corresponding entry in the Q-table by one (lines 23–25). This step populates the Q-table with the frequency of occurrence of each response key.

Finally, the algorithm returns the learning rate α, the discount factor γ, the exploration rate ϵ, the Q-table, and the Q-value (line 29). This initial setup provides the Q-Learning algorithm with basic information about the API operations and their relationships, which is further refined during testing.

B. Q-Learning-based Prioritization

In this step, ARAT-RL prioritizes API operations and selects the best parameters and value-mapping sources based on their Q-values. We present Algorithm 2 (*SelectOperation*, *SelectParameters*, and *SelectValueMappingSource*) to describe the prioritization process.

The *SelectOperation* procedure is responsible for selecting the best API operation to exercise next. The algorithm initializes variables to store the maximum average Q-value and the best operation (lines 2–3). It then iterates through each operation, calculating the average Q-value for the operation's parameters (lines 4–17). The operation with the highest average Q-value is selected as the best operation (line 15).

The *SelectParameters* procedure selects a subset of parameters for the chosen API operation. This selection is guided by the exploration rate ϵ. If a random value is greater than ϵ, the algorithm selects the top n parameters sorted by their Q-values in descending order (lines 4–8); otherwise, it randomly selects n parameters from the operation's parameters (lines 9–12). Finally, the selected parameters are returned (line 14).

The *SelectValueMappingSource* procedure is responsible for selecting the value-mapping source for the chosen API operation. The technique leverages five sources of values.

- Source 1 (example values in specification): These values are provided in the API documentation as examples for a parameter. We consider three types of OpenAPI keywords that can specify example values: `enum`, `example`, and `description` [4]. The OpenAPI documentation [4] states that users can specify example values in the *description* field, and a recent study also shows the importance of leveraging example values from descriptions [29]. However, such examples are not provided in a structured format but as natural-language text. To extract example values from the *description* field, we create a list containing each word in the text, as well as each quoted phrase.

- Source 2 (random value generated based on parameter's type, format, and constraints): This source generates random values for each parameter based on its type, format, and constraints. To generate random values, we utilize Python's built-in *random* library [30]. For date and date-time formats, we employ the *datetime* library [31] to randomly select dates and times. If the parameter has a regular expression pattern specified in the API documentation, we generate the value randomly using the *rstr* library [32]. When a minimum or maximum constraint is present, we pass it to the *random* library to ensure that the generated values adhere to the specified constraints. This approach allows ARAT-RL to explore a broader range of values compared to the example values provided in the API specification.
- Source 3 (dynamically constructed key-value pairs from request): This source extracts key-value pairs from dynamically constructed request key-value pairs. We employ Gestalt pattern matching [33] (also known as Ratcliff/Obershelp pattern recognition [34]) to identify the key most similar to the parameter name. Gestalt matching is a lightweight but effective technique that calculates the similarity between two strings by identifying the longest common subsequences and recursively comparing the remaining unmatched substrings. This technique aids in discovering producer-consumer relationships.
- Source 4 (dynamically constructed key-value pairs from response): Similar to Source 3, this source obtains key-value pairs from dynamically constructed response key-value pairs. We use the same Gestalt pattern matching approach [33] to identify the key, further assisting in the identification of producer-consumer relationships.
- Source 5 (default values): This source uses predefined default values for each data type: "string" for strings, 1.1 for numbers, 1 for integers: 1, [] for arrays, and $\{\}$ for objects. These default values can be useful for testing how the API behaves when it receives the simplest or common forms of inputs; such default values are used by other tools as well (e.g., [12]).

Similar to the *SelectParameters* procedure, the selection of the value-mapping source is guided by ϵ. If a random value is greater than ϵ, the algorithm selects the mapping source with the highest Q-value for the chosen operation (lines 4–6). This helps the algorithm focus on the most-promising mapping sources based on prior experience. Otherwise, the algorithm randomly selects a mapping source from the available sources (line 8). This randomness ensures that the algorithm occasionally explores less-promising mapping sources to avoid getting stuck in a suboptimal solution.

C. Q-Learning-based API Testing

In this step, ARAT-RL executes the selected API operations with the selected parameters and value-mapping sources, and updates the Q-values based on the response status codes. Algorithm 3 (*QLearningUpdater* and *Main*) describes the API testing process and the update of Q-values using the learning rate (α) and discount factor (γ).

The *QLearningUpdater* procedure updates the Q-values based on the response status codes. It first extracts the operation ID from the selected operation (line 2). If the response status code indicates success (2xx), the algorithm extracts key-value pairs from the request and response, assigns a reward of -1, and updates the Q-values negatively (lines 3–6). If the response status code indicates an unsuccessful request (4xx or 500), the algorithm assigns a reward of 1 and updates the Q-values positively (lines 7–10). The Q-values are updated for each parameter in the selected parameters using the Q-learning update rule (Equation 1 in §II-C (lines 11–16), and the updated Q-values are returned (line 18).

The *Main* procedure orchestrates the Q-Learning-based API testing process. It initializes the exploration rate (ϵ), its maximum value (ϵ_{max}), its adaptation factor (ϵ_{adapt}), and the time budget for testing (lines 2–4). The API specification is loaded and the Q-Learning table is initialized (lines 5–6). The algorithm then enters a loop that continues until the time limit is reached (line 7). In each iteration, the best operation, selected parameters, and selected mapping source are determined (lines 8–10). The API operation is executed with the selected parameters and mapping source, and the response is obtained (line 11). The Q-values are then updated based on the response (line 12), and the exploration rate (ϵ) is updated (line 13).

By continuously updating the Q-values based on the response status codes and adapting the exploration rate, Q-Learning-based API testing process aims to effectively explore the space of API operations and parameters.

IV. Evaluation

In this section, we present the results of empirical studies conducted to assess ARAT-RL. Our evaluation aims to address the following research questions:

1) **RQ1**: How does ARAT-RL compare with state-of-the-art REST API testing tools for in terms of code coverage?
2) **RQ2**: How does the efficiency of ARAT-RL, measured in terms of valid and fault-inducing requests generated and operations covered within a given time budget, compare to that of other REST API testing tools?
3) **RQ3**: In terms of error detection, how does ARAT-RL perform in identifying 500 responses in REST APIs compared to state-of-the-art REST API testing tools?
4) **RQ4**: How do the main components of ARAT-RL—prioritization, dynamic key-value pair construction, and sampling—contribute to its overall performance?

A. Experiment Setup

We performed our experiments using Google Cloud E2 machines, each equipped with 24-core CPU and 96 GB of RAM. We created a machine image containing all the services and tools in our benchmark. For each experiment, we deleted

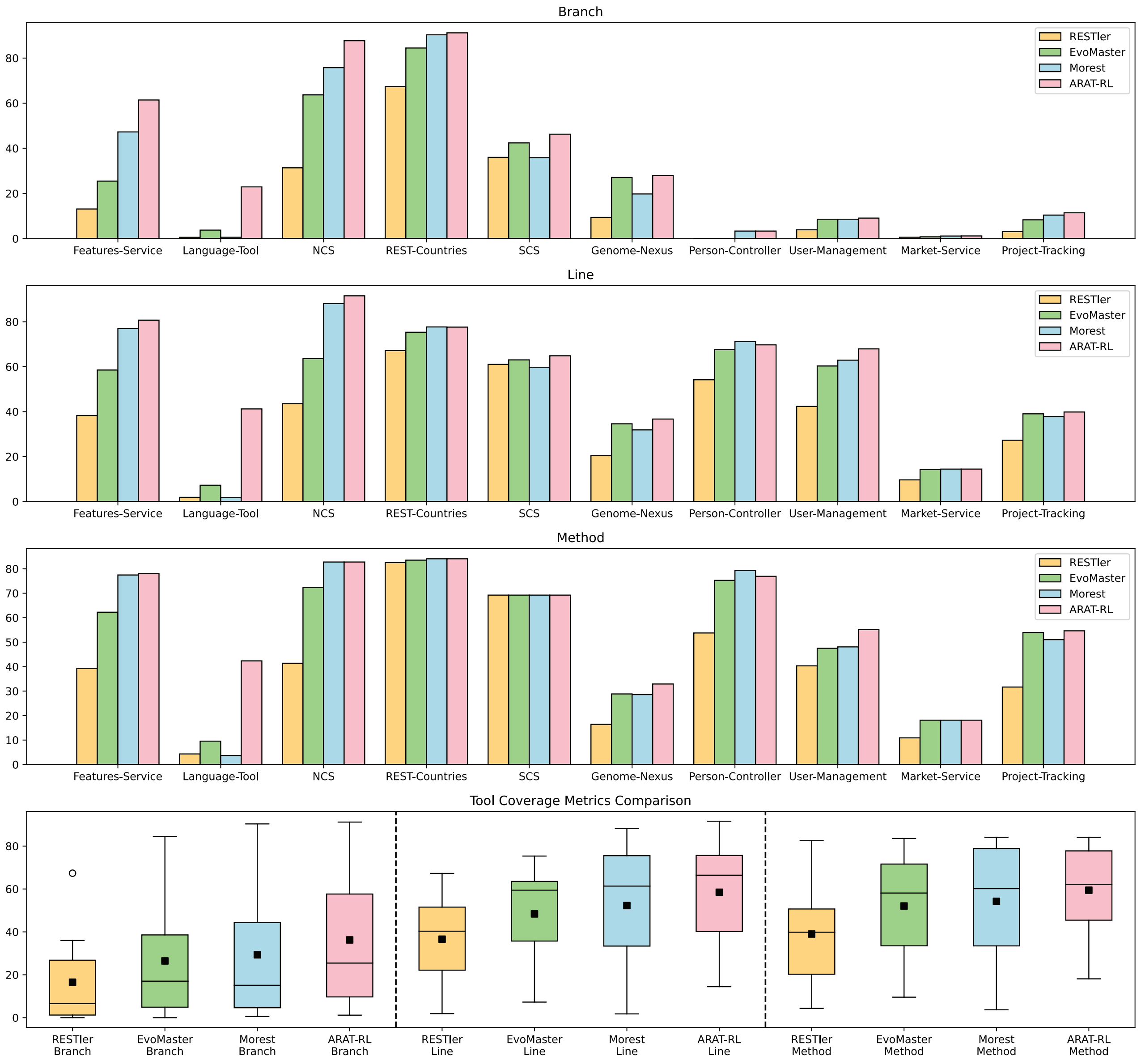

Fig. 3. Branch, line, and method coverage achieved by the tools on the benchmark services.

and recreated the machines using this image to minimize potential dependency issues. Each machine hosted all the services and tools under test, but we ran one tool at a time during the experiments. We monitored CPU and memory usage throughout the testing process to ensure that the testing tools were not affected by a lack of memory or CPU resources.

To evaluate the effectiveness and efficiency of our approach, we compared its against three state-of-the-art tools: EvoMaster [10], RESTler [12], and Morest [20]. We selected 10 RESTful services from a recent REST API testing study [18] as our benchmark. We explain the selection process of these tools and services next.

Testing Tools Selection: As a preliminary note, because ARAT-RL is a black-box approach, we considered black-box tools only in our evaluation. We believe that adding white-box tools to the evaluation would result in an unfair comparison, as these tools leverage information about the code, rather than just information in the specification, to generate test inputs.

We identified an initial set of 10 tools based on a recent study [18]. From this list, we chose (the black-box version of) EvoMaster [10] and RESTler [12]. EvoMaster employs an evolutionary algorithm for automated test case generation and was the best-performing tool in that study and in another recent comparison [35]. Its strong performance makes it an appropriate candidate for comparison. RESTler adopts a grammar-based fuzzing approach to explore APIs. It is a well-established tool in the field and, in fact, the most popular REST API testing tool in terms of GitHub stars.

Recently, two new tools have been published. Morest [20] has been shown to have superior results compared to EvoMaster. We, therefore, included Morest in our set of tools for comparison, as it could potentially outperform the other tools. The other recent tool, RestCT, was also considered for inclusion in our evaluation. However, we encountered failures while running it. We contacted the authors, who confirmed the issues and said they will work on resolving them.

RESTful Services Selection: As benchmarks for our evaluation, we selected 10 out of 20 RESTful services from a recent study [18]. We had to exclude 10 services for various reasons. Specifically, we omitted the News service, developed by the author of one of our baseline tools (EvoMaster), to avoid possible bias. Problem Controller and Spring Batch REST were excluded because they require specific domain knowledge to generate meaningful tests, so using them provides limited information about the tools. We excluded Erc20 Rest Service and Spring Boot Actuator because some APIs in these services did not provide valid responses due to external dependencies being updated without corresponding updates in the service code. Proxyprint, OCVN, and Scout API could not be included due to authentication issues that prevented them from generating meaningful responses. Finally, we excluded CatWatch and CWA Verification because of their restrictive rate limits, which slowed down the testing process and made it impossible to collect results in a reasonable amount of time.

Our final set of services consisted of Features Service, LanguageTool, NCS, REST Countries, SCS, Genome Nexus, Person Controller, User Management Microservice, Market Service, and Project Tracking System.

Result Collection: We ran each testing tool with the time budget of one hour per execution, as a previous study [18] showed that code coverage achieved by these tools tends to plateau within this duration. To accommodate randomness, we replicated the experiments ten times and calculated the average metrics across the runs.

Data collection for code coverage and status codes was done using JaCoCo [36] and Mitmproxy [37], respectively. We focused on identifying unique instances of 500 codes, which indicate server-side faults. The methodology was as follows:

1) **Stack Trace Collection:** For services that provided stack traces with 500 errors, we collected these traces, treating each unique trace as a separate fault. In the majority of cases, the errors we collected fall into this category.
2) **Response Text Analysis:** In the absence of stack traces, we analyzed the response text. After removing unrelated components (e.g., timestamps), we classified unique instances of response text linked to 500 status codes as individual faults.

This systematic approach allowed us to compile a comprehensive and unique tally of faults for our analysis.

B. RQ1: Effectiveness

To answer RQ1, we compared the tools in terms of branch, line, and method coverage achieved. Figure 3 presents the results of the study. The bar charts represent the coverage achieved by each tool for each RESTful service, whereas the boxplot at the bottom summarizes of each tool's performance on the three coverage metrics across all the services.

As the boxplot illustrates, ARAT-RL consistently outperforms the other tools in all three coverage metrics over the subject services. Looking at the performance breakdown by services (shown in the bar charts), ARAT-RL performed the best (or was tied as the best-performing tool) for all services in terms of branch coverage (with one ties), eight of the 10 services in terms of line coverage (with no tie), and nine of the 10 services in terms of method coverage (with four ties). In cases where ARAT-RL did not achieve the best results, it still achieved similar coverage rates as the best-performing tool.

ARAT-RL is especially effective when there is operation dependency, parameter dependency, and value-mapping dependency. For example, the highest coverage gains occurred for Language Tool, which has a complex set of inter-parameter and value-mapping dependencies. Meanwhile, ARAT-RL struggles with semantic parameters. For instance, its effectiveness was the lowest on Market Service, although it still matched the best-performing tool on this service. The reason for this is that the service requires input data, such as address, email, name, password, and phone number in specific formats, but ARAT-RL failed to generate valid values for these. Consequently, it was unable to create market users and then use that information for other operations in producer-consumer relationships.

On average, ARAT-RL attained 36.25% branch coverage, 58.47% line coverage, and 59.42% method coverage. In comparison, Morest, which exhibited the second-best performance, reached an average of 29.31% branch coverage, 52.27% line coverage, and 54.24% method coverage. Thus, the coverage gain of ARAT-RL over Morest is 23.69% for branch coverage, 11.87% for line coverage, and 9.55% for method coverage. EvoMaster and RESTler yield lower average coverage rates on all three metrics, with respective results of 26.45%, 48.37%, and 52.07% for EvoMaster and 16.54%, 36.58%, and 38.99% for RESTler for branch, line, and method coverage. The coverage gains of ARAT-RL compared to EvoMaster is 37.03% for branch coverage, 20.87% for line coverage, and 14.13% for method coverage; compared to RESTler, the gains are 119.17% for branch coverage, 59.83% for line coverage, and 52.42% for method coverage.

These results provide evidence that our technique can more effectively explore REST services, achieving superior code coverage, compared to existing tools, and demonstrate its potential in addressing the challenges in REST API testing.

ARAT-RL consistently outperforms RESTler, EvoMaster, and Morest in terms of branch, line, and method coverage across the subject services. However, ARAT-RL can struggle with parameters that require inputs in specific formats.

C. RQ2: Efficiency

To address RQ2, we compared ARAT-RL to Morest, EvoMaster, and RESTler in terms of the number of (1) valid

TABLE I
COMPARISON OF OPERATIONS COVERED AND VALID AND FAILURE-INDUCING REQUESTS GENERATED (2XX AND 500 STATUS CODES) BY THE TOOLS.

	ARAT-RL				Morest				EvoMaster				RESTler			
	#operations	#requests			#operations	#requests			#operations	#requests			#operations	#requests		
Service	covered	2xx+500	2xx	500	covered	2xx+500	2xx	500	covered	2xx+500	2xx	500	covered	2xx+500	2xx	500
Features Service	18	95,479	43,460	52,019	18	103,475	4,920	98,555	18	113,136	33,271	79,865	17	4,671	1,820	2,851
Language Tool	2	77,221	67,681	9,540	1	1,273	1,273	0	2	22,006	17,838	4,168	1	32,796	32,796	0
NCS	6	62,618	62,618	0	5	18,389	18,389	0	2	61,282	61,282	0	2	140	140	0
REST Countries	22	36,297	35,486	811	22	8,431	7,810	621	16	9,842	9,658	184	6	259	255	4
SCS	11	115,328	115,328	0	11	110,147	110,147	0	10	66,313	66,313	0	10	5,858	5,857	1
Genome Nexus	23	15,819	14,010	1,809	23	32,598	10,661	21,937	19	8,374	8,374	0	1	182	182	0
Person Controller	12	101,083	47,737	53,346	11	104,226	10,036	94,190	12	91,316	37,544	53,772	1	167	102	65
User Management Microservice	21	44,121	13,356	30,765	17	1,111	948	163	18	29,064	13,003	16,061	4	79	64	15
Market Service	12	29,393	6,295	23,098	6	1,399	394	1,005	5	10,697	4,302	6,395	2	1,278	0	1,278
Project Tracking System	53	23,958	21,858	2,100	42	14,906	12,904	2,002	43	15,073	13,470	1,603	3	72	65	7
Average	18	60,132	42,783	17,349	15.6	39,595	17,748	21,847	14.5	42,710	26,505	16,205	4.7	4,550	4,128	422

TABLE II
TOTAL FAULTS DETECTED BY THE TOOLS OVER 10 RUNS.

Service	RESTler	EvoMaster	Morest	ARAT-RL
Features Service	10	10	10	10
Language Tool	0	48	0	122
NCS	0	0	0	0
REST Countries	9	10	10	10
SCS	3	0	0	0
Genome Nexus	0	0	5	10
Person Controller	58	221	274	943
User Management Microservice	10	10	8	10
Market Service	10	10	10	10
Project Tracking System	10	10	10	10
Total	110	319	327	1125

and fault-inducing requests generated (as indicated by HTTP status codes 2xx and 500, respectively) and (2) operations covered within a given time budget. Although efficiency is not only dependent on these metrics, due to factors such as API response time, we feel that they still represent meaningful proxies because they indicate the extent to which the tools are exploring the API and identifying faults.

Table I shows these metrics for the 10 subject services. For each service, the table lists the number of operations covered and the number of requests made under the categories 2xx+500 (sum of 2xx code and 500 status code), 2xx, and 500, for each of the four tools. In the last row, the table presents the average number of operations covered and requests made by each tool across all the services.

In the testing time budget of one hour, ARAT-RL generated more valid and failure-inducing requests, resulting in more exploration of the testing search space. Specifically, ARAT-RL generated 60,132 valid and fault-inducing requests on average, which is 52.01% more than Morest (39,595 requests), 40.79% more than EvoMaster (42,710 requests), and 1222% more than RESTler (4,550 requests).

This difference in the number of requests can be attributed to ARAT-RL's approach of processing only a sample of key-value pairs from the response, rather than the entire response. By focusing on sampling key-value pairs, ARAT-RL efficiently identifies potential areas of improvement, contributing to a more effective REST API testing process.

Moreover, ARAT-RL covered more operations on average (18 operations) compared to Morest (15.6 operations), EvoMaster (14.5 operations), and RESTler (4.7 operations). This indicates that ARAT-RL efficiently generates more requests in a given time budget, which leads to covering more API operations, contributing to more comprehensive testing process.

Given a one-hour testing time budget, ARAT-RL generates 52.01%, 40.79%, and 1222% more valid and fault-inducing requests than Morest, EvoMaster, and RESTler, respectively. Additionally, it covers 15.38%, 24.14%, and 282.98% more operations than these tools.

D. RQ3: Fault-Detection Capability

Table II presents the total number of faults detected by each tool across 10 runs for the RESTful services in our benchmark. We note that this cumulative count might include multiple detection of the same fault in different runs. For clarity in discussion, we provide the average faults detected per run in the text below. As indicated, ARAT-RL exhibits the best fault-detection capability, uncovering on average 113 faults over the services. In comparison, Morest and EvoMaster detected 33 and 32 faults on average, respectively, whereas RESTler found the least—11 faults on average. ARAT-RL thus uncovered 9.3x, 2.5x, and 2.4x more faults than RESTler, EvoMaster, and Morest, respectively.

Comparing this data against the data on 500 response codes from Table I, we can see that, although ARAT-RL generated 20.59% fewer 500 responses, it found 250% more faults than Morest. Compared to EvoMaster, ARAT-RL generated 7.06% more 500 responses, but also detected 240% more faults. These results suggest that irrespective of whether ARAT-RL generates more or fewer error responses, it is better at uncovering more unique faults.

ARAT-RL's superior fault-detection capability is particularly evident in the Language Tool and Person Controller services, where it detected on average 12 and 94 faults, respectively. What makes this significant is that these services have larger sets of parameters. Our strategy performed well in efficiently prioritizing and testing various combinations of these parameters to reveal faults. For example, Language Tool's main operation `/check`, which checks text grammar, has 11 parameters. Similarly, Person Controller's main operation `/api/person`, which creates/modifies person instances, has eight parameters. In contrast, the other services' operations usually have three or fewer parameters.

ARAT-RL intelligently tries various parameter combinations with a reward system in Q-learning because it gives negative rewards for the parameters in the successful requests. This ability to explore various parameter combinations is a signifi-

TABLE III
RESULTS OF THE ABLATION STUDY.

	Branch	Line	Method	Faults Detected
ARAT-RL	36.25	58.47	59.42	112.10
ARAT-RL (no prioritization)	28.70 (+26.3%)	53.27 (+9.8%)	55.51 (+7%)	100.10 (+12%)
ARAT-RL (no feedback)	32.69 (+10.9%)	54.80 (+6.9%)	56.09 (+5.9%)	110.80 (+1.2%)
ARAT-RL (no sampling)	34.10 (+6.3%)	56.39 (+3.7%)	57.20 (+3.9%)	112.50 (-0.4%)

cant factor in revealing more bugs, especially in services with a larger number of parameters. These results indicate that ARAT-RL's reinforcement-learning-based approach can effectively discover faults in REST services.

ARAT-RL exhibits superior fault-detection ability, uncovering 9.3x, 2.5x, and 2.4x more bugs than RESTler, EvoMaster, and Morest, respectively. This is mainly attributed to its intelligent RL-based exploration of various parameter combinations in services with large number of parameters.

E. RQ4: Ablation Study

To address RQ4, we conducted an ablation study to assess the impact of the main novel components of ARAT-RL on its performance. We compared the performance of ARAT-RL to three variants: (1) ARAT-RL without prioritization, (2) ARAT-RL without dynamic key-value construction from feedback, and (3) ARAT-RL without sampling. Table III presents the results of this study.

As illustrated in the table, the removal of any component results in reductions in branch, line, and method coverage, as well as the number of faults detected. Eliminating the prioritization component leads to the most substantial decline in performance, with branch, line, and method coverage decreasing to 28.70%, 53.27%, and 55.51%, respectively, and the number of detected faults dropping to 100. This highlights the critical role of the prioritization mechanism in ARAT-RL's effectiveness.

The absence of feedback and sampling components also negatively affects performance. Without feedback, ARAT-RL's branch, line, and method coverage decreases to 32.69%, 54.80%, and 56.09%, respectively, and the number of found faults is reduced to 110.80. Likewise, without sampling, branch, line, and method coverage drops to 34.10%, 56.39%, and 57.20%, respectively, while the number of found faults experiences a slight increase to 112.50, which may be attributed to random variations.

These findings emphasize that each component of ARAT-RL is essential for its overall effectiveness. The prioritization mechanism, feedback loop, and sampling strategy work together to optimize the tool's performance in terms of code coverage and fault detection capabilities.

Each component of ARAT-RL–prioritization, feedback, and sampling—contributes to ARAT-RL's overall effectiveness. The prioritization mechanism, in particular, plays a significant role in enhancing ARAT-RL's performance in code coverage achieved and faults detected.

F. Threats to Validity

In this section, we address potential threats to our study's validity and the steps taken to mitigate them. Internal validity is influenced by tool implementations and configurations. To minimize these threats, we used the latest tool versions and used default options. Some testing tools might have randomness, but we addressed this issue by running each tool 10 times and computing the average results.

Our data-collection process relies on an automated script. Despite thorough testing, there remains the possibility that the script could contain errors, potentially affecting our results. To mitigate this risk, we conducted several rounds of pilot testing on a representative subset of our services before applying the script to the full study.

External validity is affected by the selection of RESTful services, the limited number of RESTful services tested, impacting the generalizability of our findings. We tried to ensure a fair evaluation by selecting a diverse set of 10 services, but we will try to have a larger and more diverse set in future work.

Construct validity concerns the metrics and tool comparisons used. Metrics such as branch, line, and method coverage, number of requests, and 500 status codes as faults, although commonly, may not capture the test tools' full quality. Additional metrics, such as mutation scores, could provide a better understanding of tool effectiveness. We measured efficiency by considering the number of meaningful requests generated by the tools, including valid and fault-inducing ones, as well as the number of operations each tool covered within a given time budget. While these metrics give us some perspective on a tool's performance, they do not consider all possible factors. Other aspects, such as the response times from the services, may also significantly impact overall efficiency.

Including more tools, services, and metrics in future studies would allow a more comprehensive evaluation of our technique's performance.

V. RELATED WORK

In this section, we provide an overview of related work in automated REST API testing, requirements based test case generation, and reinforcement learning based test case generation.

Automated REST API testing: EvoMaster [10] is a technique that offers both white-box and black-box testing modes. It uses evolutionary algorithms for test case generation, refining tests based on its fitness function and checking for 500 status code. Other black-box techniques include various tools with different strategies. RESTler [12] generates stateful test cases by inferring producer-consumer dependencies and targets internal server failures. RestTestGen [11] exploits data dependencies and uses oracles for status codes and schema compliance. QuickREST [13] is a property-based technique with stateful testing, checking non-500 status codes and schema compliance. Schemathesis [16] is a tool designed for detecting faults using five types of oracles to check response compliance in OpenAPI or GraphQL web APIs via property-based testing. RESTest [14] accounts for inter-parameter dependencies, producing nominal and faulty tests with five types of oracles. Morest [20] uses a dynamically updating RESTful-service Property Graph for meaningful test case generation. RestCT [17] employs Combinatorial Testing for RESTful API testing, generating test cases based on Swagger specifications.

Open-source black-box tools such as Dredd [38], fuzz-lightyear [39], and Tcases [40] offer various testing capabilities. Dredd [38] tests REST APIs by comparing responses with expected results, validating status codes, headers, and body content. Using Swagger for stateful fuzzing, fuzz-lightyear [39] detects vulnerabilities in micro-service ecosystems, offers customized test sequences, and alerts on unexpected successful attacks. Tcases [40] is a model-based tool that constructs an input space model from OpenAPI specifications, generating test cases covering valid input dimensions and checking response status codes for validation.

Requirements based test case generation: In requirements based testing, notable works include ucsCNL [41], which uses controlled natural language for use case specifications, and UML Collaboration Diagrams [42]. Requirements by Contracts [43] proposed a custom language for functional requirements, while SCENT-Method [44] employed a scenario-based approach with statecharts. SDL-based Test Generation [45] transformed SDL requirements into EFSMs, and RTCM [46] introduced a natural language-based framework with templates, rules, and keywords. These approaches typically deal with the goal of generating test cases as accurately as possible according to the intended requirements. In contrast, ARAT-RL focuses on more formally specified REST APIs, which provide a structured basis for testing and effectively explore and adapt during the testing process to find faults.

Reinforcement learning based test case generation: Several recent studies have investigated the use of reinforcement learning in software testing, focusing primarily on web applications and mobile apps. These challenges often arise from hidden states, whereas in our approach, we have access to all states through the API specification but face more constraints on operations, parameters, and mapping values. Zheng et al. [47] proposed an automatic web client testing approach utilizing curiosity-driven reinforcement learning. Pan et al. [48] introduced a similar curiosity-driven approach for testing Android applications. Koroglu et al. [49] presented QBE, a Q-learning-based exploration method for Android apps. Mariani et al. [50] proposed AutoBlackTest, an automatic black-box testing approach for interactive applications. Adamo et al. [51] developed a reinforcement learning-based technique specifically for Android GUI testing. Vuong and Takada [52] also applied reinforcement learning to automated testing of Android apps. Köroğlu and Sen [53] presented a method for generating functional tests from UI test scenarios using reinforcement learning for Android applications.

VI. Conclusion and Future Work

We introduced ARAT-RL, a reinforcement-learning-based approach for the automated testing of REST APIs. To assess the effectiveness, efficiency, and fault-detection capability of our approach, we compared its performance to that of three state-of-the-art testing tools. Our results show that ARAT-RLoutperformed all three tools considered in terms of branch, line, and method coverage achieved, requests generated, and faults detected. We also conducted an ablation study, which highlighted the important role played by the novel features of our technique—prioritization, dynamic key-value construction based on response data, and sampling from response data to speed up key-value construction—in enhancing ARAT-RL's performance.

In future work, we will further improve ARAT-RL's performance by addressing the limitations we observed in recognizing semantic constraints and generating inputs with specific formatting requirements (e.g., email addresses, physical addresses, or phone numbers). We plan to investigate ways to combine the RESTful-service Property Graph (RPG) approach of Morest with our feedback model to better handle complex operation dependencies. We will also study the impact of RL prioritization over time. Additionally, we plan to develop an advanced technique based on multi-agent reinforcement learning. This approach will rely on multiple agents that apply natural language analysis on the text contained in server responses, utilize a large language model to sequence operations more efficiently, and employ an enhanced value generation technique using extensive parameter datasets from Rapid API [9] and APIs-guru [8]. Lastly, we aim to detect a wider variety of bugs, including bugs that result in status codes other than 500.

Acknowledgments

We thank our anonymous reviewers for their invaluable feedback and the developers of the testing tools we employed in our empirical evaluation (EvoMasterBB, Morest, and RESTler) for making their tools accessible. This research was partially supported by NSF, under grant CCF-0725202, DARPA, under contract N66001-21-C-4024, DOE, under contract DE-FOA-0002460, and gifts from Facebook, Google, IBM Research, and Microsoft Research. This publication reflects the views only of the authors; the sponsoring agencies cannot be held responsible for such views and any use which may be made of the information contained in the paper.

Data availability

The artifact associated with this submission, which includes code, datasets, and other relevant materials, is available in our GitHub repository [21]. The artifact has been designed to reproduce the experiments presented in the paper and support further research in this domain. While we have archived the submission version of our artifact on Zenodo [54], we recommend visiting our GitHub repository to access the most recent version of our tool.

References

[1] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, Inc., 2013.

[2] S. Patni, *Pro RESTful APIs*. Springer, 2017. [Online]. Available: https://doi.org/10.1007/978-1-4842-2665-0

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

[4] The Linux Foundation, "Openapi specification," 2023. [Online]. Available: https://spec.openapis.org/oas/latest.html

[5] SmartBear Software, "Swagger," 2022. [Online]. Available: https://swagger.io/specification/v2/

[6] MuleSoft, LLC, a Salesforce company, "Raml," 2020. [Online]. Available: https://raml.org/

[7] API Blueprint, "Api blueprint," 2023. [Online]. Available: https://apiblueprint.org/

[8] APIs.guru, "Apis-guru," 2023. [Online]. Available: https://apis.guru/

[9] R Software Inc., "Rapidapi," 2023. [Online]. Available: https://rapidapi.com/terms/

[10] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, jan 2019. [Online]. Available: https://doi.org/10.1145/3293455

[11] 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*, vol. 32, 01 2022. [Online]. Available: https://doi.org/10.1002/stvr.1808

[12] 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, p. 748–758. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00083

[13] S. Karlsson, A. Čaušević, 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. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00023

[14] 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

[15] S. Karlsson, A. Čaušević, and D. Sundmark, "Automatic property-based testing of graphql apis," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2021, pp. 1–10. [Online]. Available: https://doi.org/10.1109/AST52587.2021.00009

[16] 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*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 345–346. [Online]. Available: https://doi.org/10.1145/3510454.3528637

[17] H. Wu, L. Xu, X. Niu, and C. Nie, "Combinatorial testing of restful apis," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 426–437. [Online]. Available: https://doi.org/10.1145/3510003.3510151

[18] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated test generation for rest apis: No time to rest yet," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 289–301. [Online]. Available: https://doi.org/10.1145/3533767.3534401

[19] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "A catalogue of inter-parameter dependencies in restful web apis," in *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 399–414. [Online]. Available: https://doi.org/10.1007/978-3-030-33702-5_31

[20] 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

[21] "Experiment infrastructure and data for arat-rl," 2023. [Online]. Available: https://github.com/codingsoo/ARAT-RL

[22] A. Rodriguez, "Restful web services: The basics," *IBM developerWorks*, vol. 33, no. 2008, p. 18, 2008.

[23] S. Tilkov, "A brief introduction to rest," *InfoQ, Dec*, vol. 10, 2007.

[24] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[25] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[26] "Q learning practical guide," 2023. [Online]. Available: https://ceri-num.gitbook.io/fa-paio/agir-et-apprendre-a-agir/q-learning

[27] N. Kumar, "Reinforcement learning guide with python example," 2023. [Online]. Available: https://sparkbyexamples.com/machine-learning/reinforcement-learning-in-machine-learning-with-python-example/

[28] A. Masadeh, Z. Wang, and A. E. Kamal, "Reinforcement learning exploration algorithms for energy harvesting communications systems," in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–6. [Online]. Available: https://doi.org/10.1109/ICC.2018.8422710

[29] 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, 2023, p. 1232–1243.

[30] "Python random library," 2023. [Online]. Available: https://docs.python.org/3/library/random.html

[31] "Python datetime library," 2023. [Online]. Available: https://docs.python.org/3/library/datetime.html

[32] "Rstr library," 2023. [Online]. Available: https://pypi.org/project/rstr/

[33] "Difflib sequencematcher (gestalt pattern matching)," 2023. [Online]. Available: https://docs.python.org/3/library/difflib.html

[34] P. E. Black, "Ratcliff/obershelp pattern recognition," 2021. [Online]. Available: https://www.nist.gov/dads/HTML/ratcliffObershelp.html

[35] M. Zhang and A. Arcuri, "Open problems in fuzzing restful apis: A comparison of tools," *ACM Trans. Softw. Eng. Methodol.*, may 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3597205

[36] "Jacoco," 2023. [Online]. Available: https://www.eclemma.org/jacoco/

[37] "Mitmproxy," 2023. [Online]. Available: https://mitmproxy.org/

[38] "Dredd," 2023. [Online]. Available: https://github.com/apiaryio/dredd

[39] "Fuzz-lightyear," 2023. [Online]. Available: https://github.com/Yelp/fuzz-lightyear

[40] "Tcases," 2023. [Online]. Available: https://github.com/Cornutum/tcases/tree/master/tcases-openapi

[41] F. Barros, L. Neves, E. Hori, and D. Torres, "The ucscnl: A controlled natural language for use case specifications," in *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, 01 2011, pp. 250–253.

[42] M. Badri, L. Badri, and M. Naha, "A use case driven testing process: Towards a formal approach based on uml collaboration diagrams," in *Formal Approaches to Software Testing*, A. Petrenko and A. Ulrich, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 223–235. [Online]. Available: https://doi.org/10.1007/978-3-540-24617-6_16

[43] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Requirements by contracts allow automated system testing," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, 2003, pp. 85–96. [Online]. Available: https://doi.org/10.1109/ISSRE.2003.1251033

[44] J. Ryser and M. Glinz, "A scenario-based approach to validating and testing software systems using statecharts," 1999. [Online]. Available: https://doi.org/10.5167/uzh-205008

[45] L. Tahat, B. Vaysburg, B. Korel, and A. Bader, "Requirement-based automated black-box test generation," in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, 2001, pp. 489–495. [Online]. Available: https://doi.org/10.1109/CMPSAC.2001.960658

[46] T. Yue, S. Ali, and M. Zhang, "Rtcm: A natural language based, automated, and practical test case generation framework," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 397–408. [Online]. Available: https://doi.org/10.1145/2771783.2771799

[47] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu, "Automatic web testing using curiosity-driven reinforcement learning," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 423–435. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00048

[48] 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*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 153–164. [Online]. Available: https://doi.org/10.1145/3395363.3397354

[49] 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*, 2018, pp. 105–115. [Online]. Available: https://doi.org/10.1109/ICST.2018.00020

[50] 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. [Online]. Available: https://doi.org/10.1109/ICST.2012.88

[51] 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*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 2–8. [Online]. Available: https://doi.org/10.1145/3278186.3278187

[52] 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*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 31–37. [Online]. Available: https://doi.org/10.1145/3278186.3278191

[53] Y. Köroğlu and A. Sen, "Functional test generation from ui test scenarios using reinforcement learning for android applications," *Software Testing, Verification and Reliability*, vol. 31, 10 2020. [Online]. Available: https://doi.org/10.1002/stvr.1752

[54] M. Kim, "Replication package for the paper 'adaptive rest api testing with reinforcement learning'," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8237454