A Multi-Agent Approach for REST API Testing with Semantic Graphs and LLM-Driven Inputs

Myeongsoo Kim*[1], Tyler Stennett*[2], Saurabh Sinha†[3], Alessandro Orso*[4]
*Georgia Institute of Technology, Atlanta, GA, 30332, USA.
†IBM T.J. Watson Research Center, Yorktown Heights, NY, 10598, USA.
{[1]mkim754, [2]tstennett3}@gatech.edu, [3]sinhas@us.ibm.com, [4]orso@cc.gatech.edu

***Abstract*—As modern web services increasingly rely on REST APIs, their thorough testing has become crucial. Furthermore, the advent of REST API documentation languages, such as the OpenAPI Specification, has led to the emergence of many black-box REST API testing tools. However, these tools often focus on individual test elements in isolation (e.g., APIs, parameters, values), resulting in lower coverage and less effectiveness in fault detection. To address these limitations, we present AutoRestTest, the first black-box tool to adopt a dependency-embedded multi-agent approach for REST API testing that integrates multi-agent reinforcement learning (MARL) with a semantic property dependency graph (SPDG) and Large Language Models (LLMs). Our approach treats REST API testing as a separable problem, where four agents—API, dependency, parameter, and value agents—collaborate to optimize API exploration. LLMs handle domain-specific value generation, the SPDG model simplifies the search space for dependencies using a similarity score between API operations, and MARL dynamically optimizes the agents' behavior. Our evaluation of AutoRestTest on 12 real-world REST services shows that it outperforms the four leading black-box REST API testing tools, including those assisted by RESTGPT (which generates realistic test inputs using LLMs), in terms of code coverage, operation coverage, and fault detection. Notably, AutoRestTest is the only tool able to trigger an internal server error in the Spotify service. Our ablation study illustrates that each component of AutoRestTest—the SPDG, the LLM, and the agent-learning mechanism—contributes to its overall effectiveness.**

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

I. Introduction

Modern web services increasingly depend on REST (Representational State Transfer) APIs for efficient communication and data exchange [1]. These APIs enable seamless interactions between various software systems using standard web protocols [2]. REST APIs function through common internet methods and a design that allows the server and client to operate independently [3]. The prevalence of REST APIs is evident from platforms such as APIs Guru [4] and Rapid API [5], which host extensive collections of API specifications.

In recent years, various automated testing tools for REST APIs have been developed (e.g., [6]–[22]). These tools follow a sequential process: select an operation to test, identify operations that depend on the selected operation, determine parameter combinations, and assign values to those parameters. Feedback from response status codes is then used to adjust the exploration policy at each step, either encouraging or penalizing specific choices. Although significant research has been dedicated to optimizing each individual step—operation selection, dependency identification, parameter selection, and value generation—these tools treat each step in isolation, rather than as part of a coordinated testing strategy. This isolated approach can result in suboptimal testing with a high number of invalid requests. For instance, one endpoint might benefit from extensive exploration of different parameter combinations, whereas another endpoint might require focused exploration of previously successful parameter combinations with diverse input values.

Consequently, existing tools often achieve low coverage, especially on large REST services, as shown in recent studies (e.g., Language Tool, Genome Nexus, and Market in the ARAT-RL evaluation [13], and Spotify and OhSome in the NLP2REST evaluation [23]).

To overcome these limitations, we propose AutoRestTest, a new approach that integrates a semantic property dependency graph (SPDG) and multi-agent reinforcement learning (MARL) with Large Language Models (LLMs) to enhance REST API testing. Instead of traversing all operations to find dependencies by matching input/output properties, AutoRestTest uses an embedded graph that prioritizes the properties by calculating the cosine similarity between input and output names.

Specifically, AutoRestTest employs four specialized agents to optimize the testing process. The *dependency agent* manages and utilizes the dependencies between operations identified in the SPDG, guiding the selection of dependent operations for API requests. The *operation agent* selects the next API operation to test, prioritizing operations likely to yield valuable test results based on previous results, such as successfully processed dependent operations. The *parameter agent* chooses parameter combinations for the selected operation to explore different configurations. Finally, the *value agent* manages the generation of parameter values using three data sources: values from dependent operations, LLM-generated values that satisfy specific constraints using few-shot prompting [24], and type-based random values. The value agent learns which data source is most effective for each parameter type and context.

[1]Also with AWS AI Labs, Santa Clara, CA, USA; this work was performed before the author joined AWS AI Labs.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00179

The AutoRestTest agents collaborate to optimize the testing process using the multi-agent value decomposition Q-learning approach [25], [26]. When selecting actions, each agent is employed independently using the epsilon-greedy strategy for exploitation-exploration balancing. However, during policy updates using the Q-learning equation, AutoRestTest uses value decomposition to consider the joint actions across all agents. Through centralized policy updates, each agent converges toward selecting the optimal action while accounting for the actions of the other agents.

We evaluated AutoRestTest on 12 real-world RESTful services used in previous studies [13], [23], including well-known services such as Spotify, and compared its performance with that of four state-of-the-art REST testing tools, recognized as top-performing tools in recent studies [13], [15], [27], [28]: RESTler [7], EvoMaster [29], MoRest [15], and ARAT-RL [13]. To ensure a fair comparison, we used enhanced API specifications generated by RESTGPT [30], which augments REST API documents with realistic input values generated using LLMs. To measure effectiveness, we used code coverage for the open-source services, operation coverage for the online services, and fault detection ability—measured as internal server errors triggered—for all the services. These are the most commonly used metrics in this field [27], [31].

AutoRestTest demonstrated superior performance across all coverage metrics compared to existing tools. It achieved 58.3% method coverage, 32.1% branch coverage, and 58.3% line coverage, significantly outperforming ARAT-RL, EvoMaster, RESTler, and MoRest by margins of 12–27%. For closed-source services, AutoRestTest processed 25 API operations, compared to 9–11 operations processed by the other tools. These results show that AutoRestTest can perform more comprehensive API testing than the other tools.

In terms of fault detection, AutoRestTest identified 42 operations with internal server errors, outperforming ARAT-RL (33), EvoMaster (20), MoRest (20), and RESTler (14). Notably, AutoRestTest was the only tool that detected an internal server error for Spotify [32]. We reported the errors detected on FDIC, OhSome, and Spotify, as these are actively maintained projects. The OhSome error has been confirmed and fixed [33]; we are still waiting for feedback from the developers for the other bug reports.

We also performed an ablation study, which revealed the importance of AutoRestTest's key components. Removing temporal-difference Q-learning caused the largest performance drop, with method, line, and branch coverage falling to 45.6% (-12.7), 18.2% (-13.9), and 45.8% (-12.5), respectively. SPDG removal reduced coverage to 46.7% (-11.6), 18.7% (-13.4), and 47.6% (-10.7), while LLM removal led to 47.4% (-10.9), 19.3% (-12.8), and 45.8% (-12.5). Overall, removing any component decreased coverage by 10.7–13.9%, demonstrating the significance of each component in contributing to AutoRestTest's effectiveness.

The main contributions of this work are:

- A novel REST API testing technique that reduces the operation dependency search space using a similarity-based graph model, employs multi-agent reinforcement learning to consider optimization among the testing steps, and leverages LLMs to generate realistic test inputs.
- Empirical results showing that AutoRestTest outperforms state-of-the-art REST API testing tools—even when provided with enhanced API specifications—by covering more operations, achieving higher code coverage, and triggering more failures.
- An artifact including the AutoRestTest tool, the benchmark services used in the evaluation, and detailed empirical results, which serves as a comprehensive resource for supporting further research and replication of our results [34].

```
/register:
  post:
    tags:
      - customer-rest-controller
    summary: createCustomer
    operationId: createCustomerUsingPOST
    requestBody:
      description: user
      content:
        application/json:
          schema:
            type: object
            properties:
              links:
                type: array
                items:
                  $ref: '#/components/schemas/Link'
              email:
                type: string
                maxLength: 50
                pattern: "^[\\w-]+(\\.[\\w-]+)*@([\\w-]+\\.)+[a-zA-Z]+$"
              name:
                type: string
                maxLength: 50
                pattern: "^[\\pL '-]+$"
              password:
                type: string
                maxLength: 50
                minLength: 6
                pattern: "^[a-zA-Z0-9]+$"
    responses:
      "201":
        description: Created
        content:
          '*/*':
            schema:
              $ref: '#/components/schemas/UserDTORes'
      "401":
        description: Unauthorized
      "403":
        description: Forbidden
      "404":
        description: Not Found
```

Fig. 1. A part of Market API's OpenAPI Specification.

II. Background and Motivating Example

A. REST APIs

REST APIs are a type of web APIs that conform to RESTful architectural principles [1]. These APIs enable data exchange between client and server through established protocols like HTTP [35]. The foundation of REST lies in several key concepts: statelessness, cacheability, and a uniform interface [3].

In RESTful services, clients interact with web resources by making HTTP requests. These resources represent various types of data that a client might wish to create, read, update, or delete. The API endpoint, defined by a specific resource path (e.g., /users), represents the resource that clients interact with. Different HTTP methods (e.g., POST, GET, PUT, DELETE) determine what actions can be performed on that resource, such as creating, reading, updating, or deleting data. Each unique combination of an endpoint's resource path and an

HTTP method constitutes an operation (e.g., GET /users or POST /users).

When a web service processes a request, it responds with headers, a body (if applicable), and an HTTP status code indicating the result. Successful operations typically return 2xx codes, while 4xx codes denote client-side errors, and a 500 status code indicates an internal server error.

B. OpenAPI Specification

The OpenAPI Specification (OAS) [36], which evolved from Swagger [37] in version 2 to the OAS in version 3, is a crucial standard for RESTful API design and documentation. As an industry-standard format, OAS defines the structure, functionality, and anticipated behaviors of APIs in a standardized and human-accessible way.

Figure 1 presents a portion of the Market API's OAS. This example highlights the structured approach of OAS in defining API operations. For instance, it specifies a POST request on the `/register` endpoint for creating a new customer. The request body is expected to be in `application/json` format and must include properties such as `links`, `email`, `name`, and `password`, each with specific validation constraints. The responses section defines possible outcomes, including a `201 Created` status for a successful request, along with `401 Unauthorized`, `403 Forbidden`, and `404 Not Found` status codes for various error conditions.

C. Large Language Models

Large Language Models (LLMs), like the Generative Pre-trained Transformer (GPT) series, are at the forefront of advancements in natural language processing (NLP) [38]. LLMs, trained on extensive text collections, can understand, interpret, and generate human-like text [39]. The GPT series, including GPT-3, exemplifies these advanced LLMs, demonstrating a remarkable ability to produce human-like text for various applications, from education to customer service [24], [40], [41]. Their versatility in handling diverse language-related tasks, from writing coherent text and translating languages to test case generation [42], highlights their advanced capabilities in human-like understanding [39].

D. Multi-Agent Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning where an agent learns to make decisions by interacting with its environment [43]. In this process, the agent selects actions in various situations (states), observes the outcomes (rewards), and learns to choose the best actions to maximize the cumulative reward over time. RL involves a trial-and-error approach, where the agent discovers optimal actions by experimenting with different options and adjusting its strategy based on the observed rewards. Additionally, the agent must balance between exploring new actions to gain more knowledge and exploiting known actions that provide the best reward based on its current understanding. This balance between exploration and exploitation is often controlled by parameters, such as ϵ in the ϵ-greedy strategy [43].

Multi-agent reinforcement learning (MARL) is an advanced extension of reinforcement learning that involves multiple agents interacting in a shared environment to achieve individual or collective goals [25]. MARL addresses the complexities of agent interactions, including cooperation, competition, and communication [44]. Techniques such as cooperative learning, competitive learning, and communication and coordination methods enable agents to develop optimal strategies. Applications of MARL span various domains, including autonomous driving and robotic coordination, where agents must work together or compete to optimize their performance [45]–[47]. MARL is expected to be used in areas requiring complex multi-agent decision making [48].

E. Motivating Example

Figure 1 depicts the `/register` endpoint in the Market API's OpenAPI specification. This endpoint is vital for creating user credentials needed in other components of the API. Existing REST API testing tools struggle to generate valid requests for this operation due to the strict parameter requirements: `email`, `name`, and `password` are required parameters, each with specific constraints on valid values, while `links` is an optional parameter. Moreover, these tools fail to prioritize information gained from a successful operation invocation in subsequent requests.

AutoRestTest addresses these issues through a multi-agent approach. The parameter agent employs reinforcement learning to identify valid parameter combinations, learning that (`email`, `name`, `password`) is a required parameter set while avoiding invalid combinations that would trigger 400 errors. For these parameters, the value agent first determines the appropriate data source for each parameter—choosing between LLMs, dependency values from previous operations, or random values—because different parameters require different generation strategies to create valid values. For the `/register` endpoint, it selects LLM generation as the parameters require specific formats (e.g., email format, password rules) that basic defaults cannot match, and dependency values are not available for this initial operation. It then generates context-aware values that comply with the specification's validation patterns. After a successful registration, the dependency agent utilizes the SPDG to identify operations that depend on user credentials (e.g., `/customer/cart`, `/customer/orders`) and propagates the registered user's information to these dependent operations. The operation agent then prioritizes testing these dependent operations, while the parameter and value agents reuse the strategies that were successful for the `/register` endpoint. Through value decomposition, MARL enables efficient policy updates by appropriately distributing rewards to each agent based on their contribution.

III. Our Approach

A. Overall Workflow

Figure 2 illustrates the architecture of AutoRestTest, highlighting its core components: the SPDG, the REST agents, and

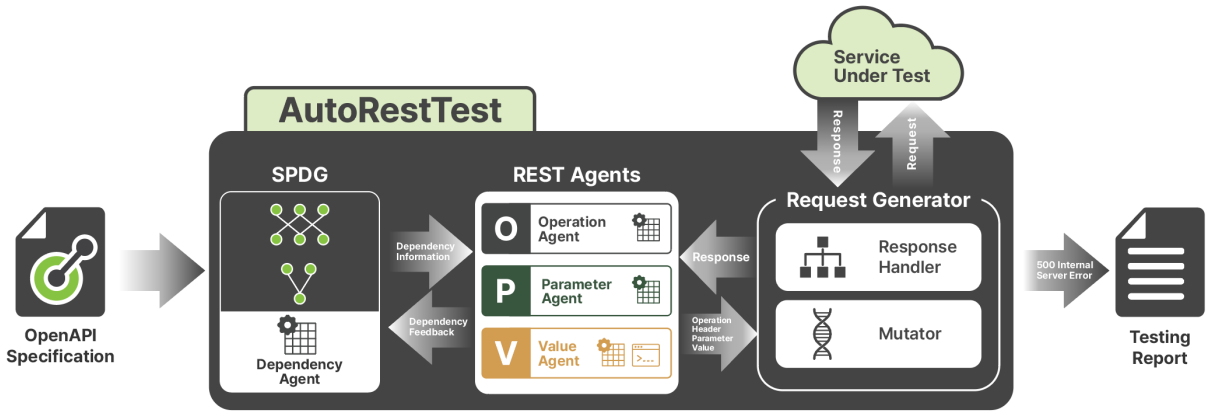

Fig. 2. Overview of our approach.

the request generator. The overall workflow consists mainly of two phases: initialization and testing execution.

1) Initialization Phase: The process begins with parsing the OpenAPI specification to extract endpoint information, parameters, and request/response schemas. Using the parsed specification, the dependency agent constructs the SPDG as a directed graph in which nodes represent API operations and edges represent potential dependencies between operations. Each weighted edge $e = (a, b)$ indicates that operation b provides values that can be used by operation a (i.e., a depends on b), with the edge weight (a value between 0 and 1) representing the semantic similarity between the operations' inputs and outputs (see Section III-C). These initial dependencies are later validated and refined through the testing process based on actual server responses.

The REST agents (operation, parameter, value, and dependency agents) are then initialized with their respective Q-tables. Each agent serves a specific purpose in the testing process: the operation agent selects API operations to test, the parameter agent determines parameter combinations, the value agent generates appropriate parameter values for each parameter, and the dependency agent manages operation dependencies from the SPDG.

2) Testing Execution Phase: The testing execution follows an iterative process. In each iteration, the operation agent first selects the next API operation based on its learned Q-values and exploration strategy. Next, the parameter agent determines which parameters to include, considering both required and optional parameters from the specification. The value agent then generates parameter values using dependencies identified by the dependency agent, LLM-generated values, or default assignments for basic parameter types. Using the SPDG, the dependency agent identifies any dependencies between the selected parameters and those used in previous operations. Finally, the request generator constructs the API request, with the mutator component modifying 20% of requests to test error handling and trigger potential 500 response codes, similar to prior work [13].

Once the request is sent to the Service Under Test (SUT) and a response is received, the response is used to update the Q-tables of all agents through reinforcement learning, refine SPDG dependencies, and store any 500 responses for the final testing report. This cycle continues until the testing time budget is exhausted. The SPDG refinement process involves increasing or decreasing edge weights, driven by rewards and penalties for dependencies, based on server responses—successful dependencies (validated by 2xx response codes) increase edge weights, whereas failed dependencies reduce edge weights, with heavily penalized edges being effectively removed from consideration. This continuous refinement helps ensure the accuracy of the dependency graph over time.

B. Q-Learning and Agent Policy

Both the SPDG and REST agent modules use the Q-learning algorithm [49] with value decomposition within their respective agents. During initialization, each agent creates a Q-table data structure that maps available actions to their expected cumulative rewards. When request generation begins, AutoRestTest addresses the two primary components of Q-learning—action selection and policy optimization—to facilitate effective communication between agents.

1) Action Selection: During action selection, agents independently choose between exploiting their best-known option or exploring new options randomly. To balance these choices, all agents follow an epsilon-greedy strategy: with probability ϵ, the agent selects a random action (exploration), and with probability $1-\epsilon$, it selects the action with the highest Q-value (exploitation), likely yielding the best results.

AutoRestTest utilizes epsilon-decay to guarantee all actions are adequately explored in its initial stages. Starting with an epsilon value of 1.0, this value decreases linearly to 0.1 over the duration of the tool's operation. This strategy, commonly used in practice, has been shown to improve performance by balancing exploration and exploitation [50].

2) Policy Optimization: After receiving a response from the server, agents update their Q-table values using the temporal-difference update rule of the Q-learning algorithm, derived from the Bellman equation [43]. This update aims to maximize the expected cumulative reward for each action taken. The Q-learning update rule is given by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta \tag{1}$$

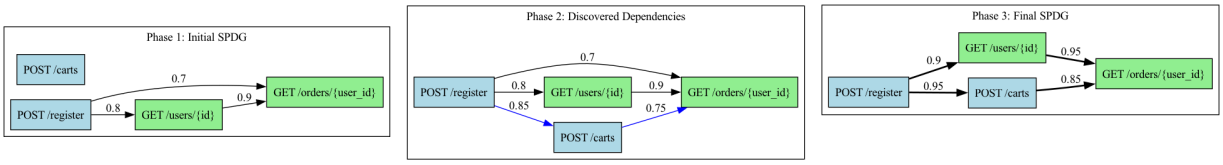

Fig. 3. Illustration of SPDG construction and refinement.

where δ represents the temporal-difference error, calculated as:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a) \tag{2}$$

where α is the learning rate, γ is the discount factor, r is the received reward, s is the current state, s' is the next state, a is the current action, and a' is an action in state s'.

Given the complexity of the multi-agent environment, AutoRestTest leverages value decomposition to optimize the joint cumulative reward, which has shown improvements for policy acquisition over independent learning. This approach assumes that the joint Q-value can be decomposed additively as follows [26]:

$$Q(s, a) = \sum_{i}^{n} Q_i(s, a_i) \tag{3}$$

where Q_i and a_i represent the Q-value and action for agent i in shared state s.

The temporal-difference error, which measures the difference between current Q-values and the optimal target Q-values, can be redefined using value decomposition to reflect the displacement from the optimal joint Q-value:

$$\delta = r + \gamma \max_{a'} \sum_{i}^{n} Q_i(s', a'_i) - \sum_{i}^{n} Q_i(s, a_i) \tag{4}$$

Using this redefined temporal-difference error, each agent updates its Q-table to converge towards the optimal joint Q-value, as depicted in the following temporal-difference equation:

$$Q_i(s, a_i) \leftarrow Q_i(s, a_i) + \alpha \Big[r + \gamma \max_{a'} \sum_{i=1}^{n} Q_i(s', a'_i) - \sum_{i=1}^{n} Q_i(s, a_i) \Big] \tag{5}$$

This value decomposition approach enables each agent to select actions independently while maintaining centralized policy updates, simplifying the implementation while enhancing coordination across agents [26].

For reward delegation, the dependency, value, and parameter agents are optimized to reward actions that generate 2xx status codes, whereas the operation agent rewards the selection of operations that generate 4xx and 5xx status codes. This balance in behavior is intended to maximize coverage by encouraging repeated attempts at creating successful requests for operations that frequently yield 4xx and 5xx status codes. For the hyperparameters α and γ, like ARAT-RL [13] and other related work [51], [52], we use values 0.1 and 0.9, respectively.

C. Semantic Property Dependency Graph

The construction of the SPDG begins with parsing the OpenAPI specification to extract information about API endpoints, parameters, and request/response schemas. As shown in Figure 3, the SPDG is initialized as a directed graph where nodes represent API operations and edges represent potential dependencies between operations based on semantic similarities.

The initialization process involves two main steps. First, for each operation in the API specification, we create a node containing the operation's ID, parameters, and response schemas. Then, we identify potential dependencies between operations by computing semantic similarities between their parameter names (inputs) and response field names (outputs) using cosine similarity with pre-trained word embeddings (e.g., GloVe [53]). When comparing two operations, if their similarity score exceeds 0.7,[1] we create an edge between them with the similarity score as its weight. To ensure all operations have some potential dependencies to explore, if an operation has no edges with scores above the threshold, we connect it to the five most similar operations.[2]

Phase 1 of Figure 3 provides an example of the initial SPDG generated by our technique. For example, the edge between `GET /users/id` and `GET /orders/user_id` has a weight of 0.9 due to the high similarity between the names of the inputs and outputs of these two operations.

1) Dependency Agent: The dependency agent manages and uses the dependencies between operations captured in the SPDG. It uses a Q-table to represent the validity of these dependencies, operating similarly to a weighted graph, where edges are assigned values that reflect confidence in each dependency. Specifically, the Q-table encodes edges from the SPDG, categorizing dependencies by parameter type (query or body) and target (parameters, body, or response). Higher Q-values on an edge indicate greater confidence in the reliability of that operation dependency, causing the agent to prioritize these relationships on future requests.

For each parameter and body property in a selected operation, the dependency agent refers to its Q-table to identify a dependent parameter, body, or response, as well as the associated operation ID. For example, as illustrated in Phase 3 of Figure 3, when testing `GET /orders/user_id`, the agent might use the value for `user_id` from a successful `GET /users/id` response, reflected by the strengthened edge weight of 0.95. The dependency agent consults AutoRestTest's tables storing successful parameters, request body properties, and

[1]We selected this value based on previous studies [23], [54].

[2]Top-five is a common threshold in top-k similarity matching [55]–[57].

decomposed responses to ensure that the selected dependency has available values. AutoRestTest recursively deconstructs response objects, allowing the dependency agent to access nested properties within response collections. Required parameters without available dependencies are fulfilled using value mappings provided by the value agent.

Although the SPDG significantly reduces the search space for operation dependencies, the reliance on semantic similarity may overlook potential candidates. To address this, the dependency agent permits random dependency queries during exploration. As shown in Phase 2 of Figure 3, if a random dependency successfully generates a valid input, AutoRestTest identifies the contributing factor and adds this new edge to the SPDG. For instance, during testing, the agent might discover that POST `/carts` returns a `cart_id` that can be used as input for GET `/orders/user_id`, leading to a new edge with weight 0.75. Similarly, when the specification contains undocumented response values (e.g., if POST `/register` returns additional user-related fields not specified in the OpenAPI document), the dependency agent evaluates these new properties for potential dependencies with other operations, as demonstrated by the edge between POST `/register` and POST `/carts` with weight 0.85 in Phase 2.

D. REST Agents

1) Operation Agent: This agent is tasked with selecting the next API operation to test. It employs reinforcement learning to prioritize operations that are likely to yield meaningful test results based on prior experiences. This agent uses a simplified state model that only tracks whether an operation is available for selection. The agent's action space encompasses all possible operations in the API specification. Each operation's Q-value is initialized to 0 and is updated based on response codes from the server. The Q-table for the operation agent stores cumulative rewards representing the proportion of unsuccessful requests for each operation in the provided service, as discussed in greater detail in the next paragraph.

The operation agent acts as the forerunner of the testing process, selecting an operation that dictates the state of the remaining agents. While the remaining agents coordinate to generate valid test cases for a given operation, the operation agent is tasked with identifying unsuccessful operations for retesting. Consequently, while the remaining agents update their Q-value using the value decomposition temporal-difference equation (Equation 5 in Section III-B), the operation agent updates its Q-values independently using a structured reward system: +2 for server errors (5xx), +1 for client errors (4xx, excluding 401 and 405), -1 for successful responses (2xx), -3 for authentication failures (401), and -10 for invalid methods (405). This reward structure encourages the agent to prioritize exploring problematic endpoints while severely penalizing systematically invalid requests and mildly penalizing endpoints with consistently successful requests.

As an example, consider the customer registration endpoint in the Market API (Figure 1). The Q-value for the endpoint is initially 0. After initial attempts at processing the operation fail, the Q-table value might increase (e.g., to 1), prompting the agent to prioritize further testing on this endpoint. Conversely, a successful test case would decrease the Q-value, directing the agent to explore other (challenging) endpoints.

2) Parameter Agent: The parameter agent is responsible for selecting parameters for the chosen API operation. It ensures that parameters used across requests are both valid and varied, covering a range of testing scenarios while addressing inter-parameter dependencies. For each operation, the parameter agent initializes a state containing the operation ID, available parameters, and required parameters, and defines its action space as possible parameter combinations. The Q-values associated with each state-action pair are initialized to 0.

Consider again the Market API's customer registration endpoint shown in Figure 1. The parameter agent initializes with the following state: $s = \{$`createCustomerUsingPOST`, [`email`, `name`, `password`, `links`], [`email`, `name`, `password`]$\}$, where the first list contains all available parameters from the request body schema, whereas the second list contains only the required parameters according to the endpoint's specification. The agent initializes a Q-table for this operation, mapping various parameter combinations to Q-values.

This setup ensures that different parameter combinations (limited to 10 by default, to account for space restrictions) are sufficiently represented in the agent's action space. The agent updates its Q-values using the value decomposition temporal-difference equation (Equation 5 in Section III-B) and the following reward structure: -1 for server errors (5xx), -2 for client errors (4xx), and +2 for successful responses (2xx).

Importantly, unused parameter combinations receive a neutral update (effectively 0) in the Q-learning process, maintaining their initial Q-values. These unused combinations are prioritized over combinations with negative rewards and deprioritized relative to those with positive rewards. For example, for the registration endpoint, if the parameter combination (`email`, `name`, `password`) consistently yields positive rewards, the unused combination (`email`, `name`, `password`, `links`) retains its initial Q-value and may be selected during exploration to test scenarios with optional parameters. Suppose that the Q-values for these parameter combinations evolved to 0.8 and 0.3, respectively. This would suggest that the inclusion of the `links` parameter is problematic and result in a lower Q-value for the configuration with all parameters.

Through this reward scheme, AutoRestTest effectively identifies valid parameter sets and addresses challenges related to undocumented inter-parameter dependency requirements, particularly in complex scenarios such as user registration, where certain parameters must be present and correctly formatted.

3) Value Agent: This agent is responsible for generating and assigning values to parameters selected by the parameter agent. For each parameter of an operation, it maintains a state containing the operation ID, parameter name, parameter type, and OpenAPI schema constraints, with its actions corresponding to possible data sources for parameter value assignment. The Q-values for each state-action pair are initialized to 0.

Consider the Market API's customer registration endpoint in Figure 1. The value agent initializes the following states for `email`, `name`, and `password` parameters:

- s = {`createCustomerUsingPOST`, `email`, `string`, {pattern: `^[\w-]+(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]+$` }}
- s = {`createCustomerUsingPOST`, `name`, `string`, {pattern: `^[\pL '-]+\$`, maxLength: 50}}
- {`createCustomerUsingPOST`, `password`, `string`, {pattern: `^[a-zA-Z0-9]+$`, minLength: 6, maxLength: 50}}

To generate a diverse set of inputs, the value agent can select inputs from the following data sources:

- *Operation Dependency Values:* When selected, the value agent collaborates with the dependency agent to map dependent values to the selected parameter. For example, the registration endpoint might reuse email addresses from prior successful registrations to test duplicate user scenarios.
- *LLM Values*: For this data source, the value agent creates (or parses if already created) values using few-shot prompting with LLMs [24].[3] For instance, the LLM might generate "john.doe@example.com" as the input value for the `email` parameter based on the specified pattern.
- *Random Values*: When this option is selected, the value agent generates random values based on the type of the selected parameter. For example, it may create a random sequence of 1-50 characters for strings, a random number between -1024 and 1024 for integers, and a random true/-false value for boolean types.

Once a request is completed with values from the chosen data source, the agent updates its Q-values based on the temporal-difference equation (Equation 5 in Section III-B), using the same reward strategy as the parameter agent (§III-D2) to refine its value generation.

For the registration endpoint, for instance, the Q-values across parameters show an average of 0.5 for LLM values, 0.2 for random values, and -0.7 for operation dependency values. In analyzing these Q-values, we observe that because the registration endpoint is required for account creation, it is less likely to derive values from operation dependencies, which results in a lower Q-value for the dependency source. Although random values are effective for simpler parameters, such as `name`, the LLM-generated values perform better for pattern-constrained fields, such as `email` and `password`.

E. Request Generator

The request generator constructs and dispatches API requests to the SUT. It works closely with the REST agents to ensure that the generated requests are both effective and comprehensive. Upon receiving responses from the SUT, the response handler processes these results and provides feedback to the REST agents, allowing refinement of future requests.

The mutator's purpose is to generate invalid requests to uncover unexpected behaviors (e.g., 500 responses). This is a crucial part of REST API testing frameworks, as state-of-the-art tools employ similar strategies such as mutating parameter types, values, and headers (e.g., using an invalid content type in the header). The mutator follows these conventions and mutates 20% of the generated requests, a strategy used by the most recent tool [13].

The request generator interacts with the REST agents to construct API requests, relying on them for detailed information about the operation to test, the parameters to use, and appropriate values for those parameters. Specifically:

- The operation agent selects the API operation to test.
- The parameter agent identifies and optimizes the parameters for the chosen operation.
- The value agent generates realistic and effective values for the parameters.

Using this information, the request generator constructs a complete API request. The request generator then dispatches the request to the SUT, which processes the request and returns a response. The response handler analyzes this response to detect any errors or unexpected behaviors. Insights from these analyses are fed back to both the REST agents and the dependency agent, allowing them to refine their strategies for future requests.

The interactions between the dependency module, the REST agents, the request generator, and the SUT establish a robust feedback loop that enhances the overall effectiveness of the testing process. This collaborative approach ensures that the generated requests are not only comprehensive but also tailored to uncover potential issues within REST APIs.

IV. Evaluation

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

1) **RQ1**: How does AutoRestTest compare with state-of-the-art REST API testing tools in terms of code coverage and operation coverage achieved?
2) **RQ2**: In terms of error detection, how does AutoRestTest perform in triggering 500 (Internal Server Error) responses compared to state-of-the-art REST API testing tools?
3) **RQ3**: How do the main components of AutoRestTest (MARL, SPDG, and LLM-based input generation) contribute to its overall performance?

A. Experiment Setup

We conducted our experiments on two cloud VMs, each equipped with a 48-core Intel(R) Xeon(R) Platinum 8260 processor with 128 GB RAM. To ensure consistent test conditions, we restarted the services and restored their databases in each testing session to eliminate potential state dependency effects across sessions. We used the default configuration and database settings for each service. We allocated dedicated resources to each service and testing tool, running them sequentially to prevent interference. Throughout the experiments, we closely monitored CPU and memory usage to ensure optimal performance without encountering resource constraints.

[3]In this work, we used GPT-3.5 Turbo with a temperature setting of 0.8 because of its known performance in REST API contexts [30].

TABLE I
REST SERVICES USED IN THE EVALUATION.

REST Service	Lines of Code	#Operations
Features Service	1688	18
Language Tool	113170	2
Rest Countries	1619	22
Genome Nexus	22143	23
Person Controller	601	12
User Management	2805	22
Market	7945	13
Project Tracking System	3784	67
YouTube-Mock	2422	1
FDIC	–	6
Spotify	–	12
Ohsome API	–	122

For our evaluation, we relied on the same set of REST API testing tools and services used by ARAT-RL [13]. Accordingly, we compared AutoRestTest with ARAT-RL [13], EvoMaster [29], MoRest [15], and RESTler [7]. Specifically, we used the latest released version or the latest commit when a release was unavailable: RESTler v9.2.4, EvoMaster v3.0.0, ARAT-RL v0.1, MoRest (obtained directly from the authors).

The ARAT-RL benchmark dataset has 10 RESTful services. In addition to these services, we included the services from the RESTGPT study [30]. Out of the total 16 services, we excluded SCS and NCS because they were written by EvoMaster's authors, and we aimed to avoid potential bias. We also excluded OCVN due to authentication issues. Lastly, we excluded OMDB, which is a toy online service with only one API operation that all testing tools can process in a second. Ultimately, we used 12 services: Features Service, Language Tool, REST Countries, Genome Nexus, Person Controller, User Management Microservice, Market Service, Project Tracking System, OhSome, YouTube-Mock, and Spotify. Table I lists the open-source services along with the lines of code and the number of API operations in each service.

For a fair comparison, because our tool utilizes LLM calls, we used the enhanced specification generated by RESTGPT, which adds realistic testing inputs to the specification using LLMs. Moreover, we used GPT-3.5-Turbo, as RESTGPT utilized this model. Based on a recent survey that describes settings and metrics for REST API testing [31], we used a one-hour time budget with ten repetitions to compute the results. To measure effectiveness and error-finding ability, we used code coverage (open-source services only), number of successfully processed operations in the specification, and number of 500 status codes, which are the most popular metrics in the literature. To collect code coverage, we used Jacoco [58]. For the number of processed operations, we used the script from the NLP2REST repository [59]. For the number of 500 status codes, we used the script available in the ARAT-RL repository [60]. This script collects 500 status codes by tracking the HTTP responses, and removes the duplicated 500 codes using the server response message for each operation.

B. RQ1: Effectiveness

The effectiveness of AutoRestTest is evaluated based on its ability to comprehensively cover more code compared to other tools. Figure 4 illustrates the line, branch, and method coverage achieved by each testing tool on the nine open-source services in our benchmark; additionally, it shows the average coverage across these APIs.

TABLE II
NUMBER OF OPERATIONS EXERCISED.

	AutoRestTest	ARAT-RL	EvoMaster	MoRest	RESTler
FDIC	6	6	6	6	6
OhSome	12	0	0	0	0
Spotify	7	5	4	4	3
Total	**25**	11	10	10	9

As shown in Figure 4, AutoRestTest outperformed the other tools in terms of method coverage, achieving 58.3% coverage on average, compared to ARAT-RL (42.1%), EvoMaster (43.1%), MoRest (31.5%), and RESTler (34.7%). This represents a significant coverage gain ranging from 15.2 to 26.8 percentage points. Similarly, AutoRestTest achieved higher line coverage, 58.3% on average, compared to the other tools, which achieved 44%, 44.1%, 33.4%, and 34.6%, respectively. Finally, in terms of branch coverage, AutoRestTest again outperformed the other tools, achieving 32.1% coverage on average compared to the other tools, which achieved 19.8%, 20.5%, 12.7%, and 11.4%, respectively.

In our evaluation, we also measured the number of processed operations for online services for which source code is unavailable: OhSome and Spotify. Table II presents these results, which highlight AutoRestTest's ability to handle a larger number of operations. Specifically, AutoRestTest exercised 25 operations in total, compared to ARAT-RL, EvoMaster, MoRest, and RESTler, which processed 11, 10, 10, and 9 operations, respectively. These results on achieved code coverage and successfully exercised operations demonstrate AutoRestTest's effectiveness on a range of different REST APIs and how it improves on the state of the art.

In most cases, there were notable performance gains in Genome Nexus, Person Controller, User Management, Market, YouTube, OhSome, and Spotify. Conversely, for Features Service, REST Countries, and Project Tracking System, AutoRestTest's results did not show much difference compared to the second-best performing tool in our set. These four services have a notable characteristic in common: the number of input parameters in their APIs is mostly 1 to 2, and the services are therefore easier to test. This result shows that AutoRestTest can effectively explore REST APIs, especially for services with a large search space.

> AutoRestTest achieves considerable gains in code coverage, with method coverage increasing between 15.2 to 26.8 percentage points, line coverage between 14.2 and 24.8 percentage points, and branch coverage between 11.6 and 20.7 percentage points compared to the other tools considered. The improvement in performance is particularly noticeable on large and complex services with many input parameters.

Fig. 4. Comparison of code coverage metrics across tools and services: line, branch, and method coverage.

TABLE III
SERVICE FAILURES TRIGGERED (500 RESPONSE CODES).

REST APIs	**AutoRestTest**	ARAT-RL	EvoMaster	MoRest	RESTler
Features Service	1	1	1	1	1
Language Tool	1	1	1	0	0
REST Countries	1	1	1	1	1
Genome Nexus	1	1	0	1	0
Person Contoller	8	8	8	8	3
User Management	1	1	1	1	1
Market	1	1	1	1	1
Project Tracking System	1	1	1	1	1
YouTube	1	1	1	1	1
FDIC	6	6	6	6	6
OhSome	20	12	0	0	0
Spotify	1	0	0	0	0
Total	**42**	33	20	20	14

C. RQ2: Fault Detection Capability

We evaluated the fault detection capability of AutoRestTest by counting how many 500 Internal Server Errors it identified. Table III shows the number of such errors detected by AutoRestTest, ARAT-RL, EvoMaster, MoRest, and RESTler. As the data in the table show, AutoRestTest detected a total of 42 500 Internal Server Errors across the evaluated REST APIs, far outperforming the other tools on this metric. ARAT-RL detected 33 errors, EvoMaster and MoRest detected 20 errors each, and RESTler detected 14 errors.

Specifically, AutoRestTest identified significantly more errors in the OhSome service (20 errors) compared to ARAT-RL (12 errors), with none detected by EvoMaster, MoRest, or RESTler. Additionally, AutoRestTest was the only tool to detect an error in the Spotify service. It is important to note that both OhSome and Spotify are active services; Spotify, for example, has 615 million users, and the OhSome service has recent GitHub commits. We reported the detected issues, and the OhSome errors were accepted, whereas we are still waiting for Spotify's response [32], [33]. This improvement in fault detection is somehow expected, as AutoRestTest achieves the highest coverage among the other tools, which is strongly correlated with fault-finding ability in REST API testing [27].

To illustrate the utility of AutoRestTest's specific components in fault detection on a specific example, consider the following sequence of operations in the Ohsome service, which shows the capabilities of the SPDG. AutoRestTest begins by successfully querying the POST `/elements/area/ratio` endpoint from the Ohsome service with its `filter2` parameter assigned to `node:relation`. Subsequently, AutoRestTest targets the GET `/users/count/groupBy/key` endpoint, where the dependency agent applies the SPDG's semantically-created dependency edges to identify a potential connection between the `filter` parameter of the new operation and the `filter2` parameter of the previous operation. When the dependency agent reuses the `node:relation` value from the previously successful `filter2` parameter in the new request, the SPDG uncovers an unexpected 500 Internal Server Error. Typically, an invalid `filter` value would trigger a client error, but this server-side error indicates that the SPDG identified a deeper, unanticipated fault in the server's value handling. The other tools overlook the correlation between the two parameters due to either the minor naming differences or improper dependency modeling, and fail to expose this error.

For another example, AutoRestTest shows the effectiveness of its LLM value generation in the interactions with the Spotify API. The GET `/playlists/playlist_id/tracks`

TABLE IV
CODE COVERAGE ACHIEVED BY DIFFERENT TOOL VARIANTS.

	Method	Line	Branch
AutoRestTest	58.3%	32.1%	58.3%
1. Without LLM	47.4% (-10.9%)	19.3% (-12.8%)	45.8% (-12.5%)
2. Without Learning	45.6% (-12.7%)	18.2% (-13.9%)	45.8% (-12.5%)
3. Without SPDG	46.7% (-11.6%)	18.7% (-13.4%)	47.6% (-10.7%)

operation in Spotify's API requires specific knowledge regarding Spotify's `playlist_id` formation. Spotify generates Spotify IDs for playlists that are typically 22 characters long with constraints on the permitted letters and patterns. Where many tools fail to create valid IDs, AutoRestTest's value agent leverages its LLM to generate valid Spotify IDs for playlists. AutoRestTest is thus able to successfully query the GET `/playlists/playlist_id/tracks` operation, with rippling effects across the service. For instance, after retrieving the International Standard Recording Code (ISRC) from the playlist's tracks, AutoRestTest's mutator randomly selects an ISRC to use as the `user_id` in the subsequent GET `/users/user_id/playlists` operation. This sequence reveals a hidden dependency conflict that results in a 500 Internal Server Error. Other tools fail to exercise the GET `/playlists/playlist_id/tracks` operation entirely and are hence unable to locate this error.

> AutoRestTest outperforms the other tools in terms of fault detection capability. It identifies a total of 42 instances of 500 Internal Server Errors, whereas ARAT-RL, EvoMaster, MoRest, and RESTler detected 33, 20, 20, and 14 errors, respectively.

D. RQ3: Ablation Study

To understand the contribution of each component in AutoRestTest, we conducted an ablation study in which we removed specific elements of the approach: the LLM-based input generation, the agent learning step in MARL, and the SPDG. Because our tool heavily depends on agents, it was not feasible to create a reasonable tool without MARL entirely. Therefore, instead of removing all the agents, we only removed the temporal-difference Q-learning from the agents. Table IV presents the impact of these removals on method, line, and branch coverage.

Removing the temporal-difference Q-learning leads to the most significant decrease in overall metrics, dropping the coverage rates to 45.6%, 18.2%, and 45.9% for method, line, and branch coverage. The learning step's contribution is crucial in optimizing the testing process through strategic exploration and exploitation of testing paths. Without the multi-agent learning step, the tool repeated the same requests and failed to properly update its agents with feedback.

The removal of the SPDG has the next most significant impact in terms of method and line coverage, dropping the performance significantly to 46.7%, 18.7%, and 47.6% for method, line, and branch coverage. This result indicates that the SPDG plays a critical role in identifying dependencies and guiding test generation by reducing the search space, thus helping identify the dependent API operations.

Removing the LLM alone also leads to a substantial decrease in performance, with method, line, and branch coverage dropping to 47.4%, 19.3%, and 45.8%. The primary reason for this difference is the LLM's ability to generate diverse and appropriate test inputs. These inputs are essential for exercising the API, as they help uncover various parameter and operation dependencies. Furthermore, operation and parameter dependencies cannot be accurately identified without resolving parameter constraints when they exist.

Notably, without the SPDG, AutoRestTest exercised only 5 operations for Spotify, whereas with the SPDG, it consistently covered 7 operations.

> The ablation study shows that each component of AutoRestTest (LLM, MARL, and SPDG) contributes considerably to the effectiveness of the approach, and removing any component drops the coverage significantly. The decrease in method, line, and branch coverage ranges, in percentage points, between 10.9 and 12.7, 12.8 and 13.9, and 10.7 and 12.5, indicating that each component plays an important role.

E. Threats to Validity

Like for any empirical study, there are potential threats to the validity of our results. Regarding construct validity, the use of ChatGPT-3.5-Turbo as our LLM component [39] introduces potential data leakage, as it may have been trained on API-related content, potentially affecting our results. While our ablation study demonstrates the LLM's importance, this limitation should be considered when interpreting our findings. Additionally, technical choices like the 20% mutation rate [13] and ChatGPT's default parameters [30] may affect result comparability.

REST APIs' inherently flaky behavior (e.g., due to network issues) introduces possible threats of internal validity. To mitigate this issue, we performed multiple test runs and averaged the results. We also carefully inspected our code and results to mitigate the risk of implementation errors.

The uneven quality of OpenAPI specifications [61] can also affect the validity of our results. While this is an inherent and somehow unavoidable issue, we have left to future work the investigation of the impact of specification completeness and accuracy on AutoRestTest's performance.

Finally, our selection of REST APIs benchmarks can affect external validity. While we used a diverse set of real-world APIs, AutoRestTest may perform differently on different benchmarks. The availability of our dataset and code will allow other researchers to validate and extend our evaluation.

V. RELATED WORK

A. Automated REST API Testing

Automated testing for REST APIs has employed various strategies. EvoMaster [29] uses both white-box and black-box techniques and applies evolutionary algorithms to refine

test cases, focusing on detecting server errors. RESTler [7] generates stateful tests by inferring producer-consumer dependencies, aiming to uncover server failures. Tools such as RestTestGen [6] exploit data dependencies and utilize oracles to validate status codes and response schemas. MoRest [15] adopts model-based testing to simulate user interactions and generate test cases, while RestCT [12] employs combinatorial testing techniques to explore parameter value combinations systematically. ARAT-RL [13] introduces reinforcement learning to adapt and refine API testing strategies based on real-time feedback. Techniques such as QuickREST [8], Schemathesis [11], and RESTest [9] use property-based testing and various oracles to ensure compliance with OpenAPI or GraphQL specifications. Tools such as Dredd [62], fuzz-lightyear [63], and Tcases [14] provide diverse testing capabilities, from validating responses against expected results to detecting vulnerabilities and validating Swagger-based specifications.

The closest approach to AutoRestTest that incorporates reinforcement learning is ARAT-RL. However, ARAT-RL does not employ a comprehensive model to represent operation dependencies, which reduces its effectiveness. Given the difficulties in embedding potential dependencies into the action space of an agent, ARAT-RL considers weighted probabilities using parameter frequency to guide potential dependencies. This can result in the inefficient prioritization of unrelated operations in subsequent requests. Unlike ARAT-RL, AutoRestTest uses the SPDG to narrow the dependency search.

MoRest, with its RESTful Property Graph (RPG), is the closest approach to AutoRestTest in terms of using a graph model. However, MoRest is incapable of progressively adapting its requests according to server feedback, rendering its dependency sequences less effective.

Concurrently to this work, two additional REST API testing techniques were proposed: DeepREST [17] and LlamaRestTest [22]. DeepREST uses deep reinforcement learning to discover implicit API constraints, employing a single agent that learns operation orderings through a reward mechanism. However, deep learning's black-box nature makes it difficult to track how and why specific testing decisions are made. In contrast, AutoRestTest's reward mechanism is more effective because uses multiple specialized agents and is able to track how each agent's decisions contribute to the overall testing strategy. Additionally, while DeepREST learns dependencies purely through trial and error, AutoRestTest reduces the search space upfront using the SPDG. LlamaRestTest fine-tunes small language models specifically for REST API testing tasks, using Llama models [64] to identify inter-parameter dependencies and generate valid inputs. LlamaRestTest mainly focuses on LLM-driven testing, whereas our approach uses LLMs only to generate parameter values while relying on the SPDG for efficient dependency identification and multi-agent reinforcement learning for dynamic optimization across all testing steps. We could not compare with either of these approaches in our evaluation because they had not yet been published when we performed this work.

B. LLM-based REST API Testing and Analysis

Recent advancements in LLMs have resulted in improved REST API testing and analysis approaches. NESTFUL [65] provides a benchmark for evaluating LLMs on nested API calls, revealing challenges in handling complex API interactions. RESTSpecIT [66] demonstrates automated specification inference and black-box testing with minimal user input, while RestGPT [67] introduces coarse-to-fine online planning for improved task decomposition and API selection. While these approaches primarily focus on REST API analysis, AutoRestTest focuses on testing. RESTSpecIT includes some testing capabilities, but it is limited to basic parameter mutation and lacks advanced testing features such as operation/parameter dependency identification.

C. Reinforcement Learning-Based Test Case Generation

Recent studies have explored reinforcement learning for software testing, particularly for web and mobile applications. For instance, Zheng and colleagues proposed a curiosity-driven reinforcement learning approach for web client testing [68], and Pan and colleagues applied a similar technique for Android application testing [69]. Other work includes QBE, a Q-learning-based exploration method for Android apps [70], and AutoBlackTest, an automatic black-box testing approach for interactive applications [71]. Reinforcement learning has also been used specifically for Android GUI testing [72]–[74]. While these approaches use single-agent-based techniques, AutoRestTest decomposes the test-generation problem into multiple components (operation, parameter, value, dependency) and uses a multi-agent reinforcement algorithm to reward the different components' contributions in each reinforcement learning step. Additionally, AutoRestTest introduces the SPDG to reduce the search space of operation dependency.

VI. CONCLUSION AND FUTURE WORK

In this paper, we introduced AutoRestTest, a new technique that leverages multi-agent reinforcement learning, the semantic property dependency graph, and large language models to enhance REST API testing. By optimizing specialized agents for dependencies, operations, parameters, and values, AutoRestTest addresses the limitations of existing techniques and tools. Our evaluation on 12 state-of-the-art REST services shows that AutoRestTest can significantly outperform leading REST API testing tools in terms of code coverage, operation coverage, and fault detection. Furthermore, our ablation study confirms the individual contributions of the MARL, LLMs, and SPDG components to the tool's effectiveness. In future work, we will explore the dynamic adaptation of testing strategies, optimize performance and scalability (e.g., through fine-tuning LLMs), and develop more sophisticated fault-classification approaches.

ACKNOWLEDGMENTS

This work was partially supported by NSF, under grant CCF-0725202 and DOE, under contract DE-FOA-0002460, and gifts from Facebook, Google, IBM Research, and Microsoft Research.

REFERENCES

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

[2] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big" web services: Making the right architectural decision," in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 805–814. [Online]. Available: https://doi.org/10.1145/1367497.1367606

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

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

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

[6] 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

[7] 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

[8] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 131–141.

[9] 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

[10] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 1–10. [Online]. Available: https://doi.org/10.1109/AST52587.2021.00009

[11] 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

[12] 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

[13] M. Kim, S. Sinha, and A. Orso, "Adaptive rest api testing with reinforcement learning," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 446–458. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00218

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

[15] 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

[16] D. Corradini, M. Pasqua, and M. Ceccato, "Automated black-box testing of mass assignment vulnerabilities in restful apis," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2553–2564.

[17] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato, "Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1383–1394. [Online]. Available: https://doi.org/10.1145/3691620.3695511

[18] A. Arcuri, M. Zhang, S. Seran, J. P. Galeotti, A. Golmohammadi, O. Duman, A. Aldasoro, and H. Ghianni, "Tool report: Evomaster—black and white box search-based fuzzing for rest, graphql and rpc apis," *Automated Software Engineering*, vol. 32, no. 1, pp. 1–11, 2025.

[19] T. Le, T. Tran, D. Cao, V. Le, T. N. Nguyen, and V. Nguyen, "Kat: Dependency-aware automated api testing with large language models," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2024, pp. 82–92.

[20] J. Chen, Y. Chen, Z. Pan, Y. Chen, Y. Li, Y. Li, M. Zhang, and Y. Shen, "Dyner: Optimized test case generation for representational state transfer (rest) ful application programming interface (api) fuzzers guided by dynamic error responses," *Electronics*, vol. 13, no. 17, p. 3476, 2024.

[21] T. Stennett, M. Kim, S. Sinha, and A. Orso, "Autoresttest: A tool for automated rest api testing using llms and marl," 2025. [Online]. Available: https://arxiv.org/abs/2501.08600

[22] M. Kim, S. Sinha, and A. Orso, "Llamaresttest: Effective rest api testing with small language models," 2025. [Online]. Available: https://arxiv.org/abs/2501.08598

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

[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[25] K. Zhang, Z. Yang, and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," *Handbook of reinforcement learning and control*, pp. 321–384, 2021.

[26] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, and T. Graepel, "Value-decomposition networks for cooperative multi-agent learning," *arXiv preprint arXiv:1706.05296*, 2017.

[27] 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

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

[29] 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

[30] M. Kim, T. Stennett, D. Shah, S. Sinha, and A. Orso, "Leveraging large language models to improve rest api testing," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 37–41. [Online]. Available: https://doi.org/10.1145/3639476.3639769

[31] A. Golmohammadi, M. Zhang, and A. Arcuri, "Testing restful apis: A survey," *ACM Trans. Softw. Eng. Methodol.*, aug 2023. [Online]. Available: https://doi.org/10.1145/3617175

[32] "Error report for spotify," 2024. [Online]. Available: https://community.spotify.com/t5/Spotify-for-Developers/500-or-502-Error-on-GET-users-user-id-playlists-Operation/m-p/6110695#M14091

[33] Anonymous, "Error report for ohsome," 2024. [Online]. Available: https://github.com/GIScience/ohsome-api/issues/327

[34] SE@GT, "Experiment infrastructure, data, and results for autoresttest," https://github.com/selab-gatech/AutoRestTest/, 2024.

[35] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext transfer protocol–http/1.0," MIT/LCS, Tech. Rep., 1996. [Online]. Available: https://www.ietf.org/rfc/rfc1945.txt

[36] OpenAPI, "Openapi standard," https://www.openapis.org, 2023.

[37] S. Software, "Swagger," https://swagger.io/specification/v2/, 2023.

[38] M. U. Hadi, R. Qureshi, A. Shah, M. Irfan, A. Zafar, M. B. Shaikh, N. Akhtar, J. Wu, S. Mirjalili *et al.*, "Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects," 2023.

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

[40] D. Baidoo-Anu and L. O. Ansah, "Education in the era of generative artificial intelligence (ai): Understanding the potential benefits of chatgpt in promoting teaching and learning," *Journal of AI*, vol. 7, no. 1, pp. 52–62, 2023.

[41] P. A. C. Debby R. E. Cotton and J. R. Shipway, "Chatting and cheating: Ensuring academic integrity in the era of chatgpt," *Innovations in Education and Teaching International*, vol. 61, no. 2, pp. 228–239, 2024. [Online]. Available: https://doi.org/10.1080/14703297.2023.2190148

[42] R. Pan, M. Kim, R. Krishna, R. Pavuluri, and S. Sinha, "Multi-language unit test generation using llms," *arXiv preprint arXiv:2409.03093*, 2024.

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

[44] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "A survey and critique of multiagent deep reinforcement learning," *Autonomous Agents and Multi-Agent Systems*, vol. 33, no. 6, pp. 750–797, 2019.

[45] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1610.03295*, 2016.

[46] L. Busoniu, R. Babuska, and B. De Schutter, "A comprehensive survey of multiagent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 2, pp. 156–172, 2008.

[47] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[48] Y. Yang and J. Wang, "An overview of multi-agent reinforcement learning from game theoretical perspective," *arXiv preprint arXiv:2011.00583*, 2020.

[49] 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

[50] V. Kumar and M. Webster, "Importance sampling based exploration in q learning," *arXiv preprint arXiv:2107.00602*, 2021.

[51] gibberblot, "gibberblot's blog post," https://gibberblot.github.io/rl-notes/single-agent/temporal-difference-learning.html, 2024.

[52] M. Kim, S. Pande, and A. Orso, "Improving program debloating with 1-du chain minimality," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 384–385. [Online]. Available: https://doi.org/10.1145/3639478.3643518

[53] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[54] N. Rekabsaz, M. Lupu, and A. Hanbury, "Exploration of a threshold for similarity based on uncertainty in word embedding," in *European Conference on Information Retrieval*. Springer, 2017, pp. 396–409.

[55] K. Komiya, Y. Abe, H. Morita, and Y. Kotani, "Question answering system using q & a site corpus query expansion and answer candidate evaluation," *SpringerPlus*, vol. 2, pp. 1–11, 2013.

[56] J. Yao, C. Yuan, X. Li, Y. Wang, and Y. Su, "Beyond top-k: knowledge reasoning for multi-answer temporal questions based on revalidation framework," *PeerJ Computer Science*, vol. 9, p. e1725, 2023.

[57] M. Li, X. Shen, Y. Sun, W. Zhang, J. Nan, D. Gao *et al.*, "Using semantic text similarity calculation for question matching in a rheumatoid arthritis question-answering system," *Quantitative Imaging in Medicine and Surgery*, vol. 13, no. 4, p. 2183, 2023.

[58] E. Team, "Jacoco," https://www.eclemma.org/jacoco/, 2023.

[59] M. Kim and D. Corradini, "Experiment infrastructure, data, and results for nlp2rest," https://github.com/codingsoo/nlp2rest, 2024.

[60] M. Kim", "Experiment infrastructure and data for arat-rl," 2024. [Online]. Available: https://github.com/codingsoo/ARAT-RL

[61] R. Huang, M. Motwani, I. Martinez, and A. Orso, "Generating rest api specifications through static analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639137

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

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

[64] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[65] K. Basu, I. Abdelaziz, K. Bradford, M. Crouse, K. Kate, S. Kumaravel, S. Goyal, A. Munawar, Y. Rizk, X. Wang *et al.*, "Nestful: A benchmark for evaluating llms on nested sequences of api calls," *arXiv preprint arXiv:2409.03797*, 2024.

[66] A. Decrop, G. Perrouin, M. Papadakis, X. Devroey, and P.-Y. Schobbens, "You can rest now: Automated specification inference and black-box testing of restful apis with large language models," *arXiv preprint arXiv:2402.05102*, 2024.

[67] Y. Song, W. Xiong, D. Zhu, W. Wu, H. Qian, M. Song, H. Huang, C. Li, K. Wang, R. Yao *et al.*, "Restgpt: Connecting large language models with real-world restful apis," *arXiv preprint arXiv:2306.06624*, 2023.

[68] 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

[69] 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

[70] 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*. Los Alamitos, CA, USA: IEEE Computer Society, 2018, pp. 105–115. [Online]. Available: https://doi.org/10.1109/ICST.2018.00020

[71] 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*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 81–90. [Online]. Available: https://doi.org/10.1109/ICST.2012.88

[72] 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

[73] 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

[74] 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