CRUDinfer: Automated CRUD Semantics Inference for REST APIs Through Black-box Testing

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

Davide Corradini*
University of Luxembourg
Luxembourg, Luxembourg
davide.corradini@uni.lu

Michele Perlotto
University of Naples Federico II
Naples, Italy
michele.perlotto@unina.it

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

Abstract

REST APIs are the de facto standard for web service interaction, praised for flexibility and simplicity of adoption. Nevertheless, the lack of mandatory implementation guidelines for REST APIs has led to the diffusion of poor-quality and difficult-to-maintain web services. A key concern in this context is the misuse of CRUD (Create, Read, Update, and Delete) semantics of API operations. While best practices suggest mapping CRUD verbs to HTTP methods (`POST`, `GET`, `PUT/PATCH`, and `DELETE`, respectively), many implementations fail to adhere to such a guideline. This common *anti-pattern* makes interaction with the API ambiguous, hindering maintainability and decreasing the effectiveness of automated REST API testing.

In this paper, we propose CRUDinfer, a novel approach to automatically infer REST API operation CRUD semantics by leveraging (black-box) interactions with the API. The approach incrementally refines the knowledge about API operations' CRUD semantics via *CRUD test scenarios*. Specifically, it employs interaction patterns typical of each CRUD semantics verb to craft test scenarios (i.e., HTTP interactions) for API operations with the aim of confirming their semantics. Testing failures indicate a mismatch between the intended CRUD semantics and the actual implementation. Thus, it refines API operations' CRUD semantics knowledge through improved test scenarios. Empirical evaluation indicates high inference capabilities for CRUDinfer, with an overall precision higher than 95% on the considered benchmark REST APIs.

CCS Concepts

• **Information systems** → **RESTful web services**; • **Software and its engineering** → *Maintaining software*; Software defect analysis.

*Affiliated with the University of Verona when this work was done.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3787826

Keywords

REST APIs, OpenAPI specification, CRUD semantics inference, Black-box software analysis

ACM Reference Format:
Michele Pasqua, Davide Corradini, Michele Perlotto, and Mariano Ceccato. 2026. CRUDinfer: Automated CRUD Semantics Inference for REST APIs Through Black-box Testing. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3787826

1 Introduction

REST APIs have become the dominant standard for remote access and manipulation of resources provided by web services. This architectural style, introduced in the early 2000s by Roy Fielding [14], has completely changed the way web and mobile applications interact with data. The spread of REST APIs has been rapid and global, making them ubiquitous in modern software development. Golmohammadi et al. [18] acknowledges the industry's widespread adoption of RESTful APIs as the foundational choice for modern applications, particularly in microservices and cloud environments. Indeed, *RapidAPI* [47] lists more than 90 thousand Web APIs, where the most common are RESTful ones.

One of the main advantages of REST APIs lies in their ease of implementation, made possible by the adoption of consolidated technologies and standards, such as the use of the HTTP protocol to manage requests and responses, along with the use of formats like JSON for data representation. Pautasso et al. [44] remark that adopting the REST architectural style avoids the need for complex and specialized infrastructure, development frameworks, and definition languages, thus reducing the development effort and making implementation quicker and easier. This ease of implementation has, however, allowed developers with any level of experience to create and release REST-based applications. Moreover, the fact that REST is an architectural style, and not a protocol with mandatory implementation directives, has led to the diffusion of REST APIs not adhering to consolidated development best practices, hindering interoperability and maintenance of REST-based applications.

A prominent example is the misuse of HTTP methods with wrong CRUD semantics for REST API operations. An API operation is an exposed functionality consisting of an HTTP method

and a path. The *Create*, *Read*, *Update*, and *Delete* (CRUD) semantics refer to the expected behavior of an API operation, which should create, read, update, or delete resources. Best practices suggest mapping CRUD semantics verbs to API operation HTTP methods (`POST`, `GET`, `PUT/PATCH`, and `DELETE`, respectively). This principle, advocated by Fielding [14] and supported by common guidelines to adopt when developing REST services [44], limits the risk of incorrect operations implementation, avoiding unexpected API behavior. Rodriguez et al. [48] argue that the failure to correctly use HTTP methods introduces ambiguity and breaks the widely understood client-API contract. This ambiguity increases the likelihood of client and intermediary components misinterpreting the API's intent, thereby leading to unexpected API behavior in a distributed system. CRUD principle adherence also allows rapid use of APIs to exercise the required functionalities, thus reducing development time and effort, and it facilitates code maintenance, ensuring that CRUD operations are managed reliably during software evolution. Indeed, Google Cloud API Design Guide [20] emphasizes that using standard HTTP methods for standard actions (the CRUD mapping) ensures API consistency and usability. This consistency allows developers to quickly understand and integrate new APIs, lowering the cognitive load and, consequently, reducing overall development time/effort for client implementations. Similarly, Microsoft Azure REST API Design Guide [35] enforces the correct use of HTTP methods for resource manipulation. They stress that using standard methods for CRUD operations is crucial for consistency, predictability, and maintainability. Consistency across a large ecosystem ensures that developer tools, documentation, and client-side code can be managed reliably as the services evolve.

Knowledge of CRUD semantics of API operations is also crucial for automated REST API black-box testing tools, which aim to automatically verify the correctness and security of an API by leveraging its formal documentation (e.g., its OpenAPI specification). In order to correctly craft test interactions, such tools must necessarily know what API operations are supposed to do (e.g., create a resource or read a list of resources). In the absence of auxiliary information, testing tools typically assume that the HTTP method matches the operation CRUD semantics (e.g., an operation creating a resource is expected to use the `POST` method, while an operation reading a list of resources is expected to use the `GET` method).

Not surprisingly, many API implementations fail to adhere to the aforementioned best practice. CRUD semantics mismatch is indeed a common *anti-pattern* in the context of REST APIs, hindering the maintainability of web services and decreasing the effectiveness of automated REST API testing. Rodriguez et al. [48] also performed a large-scale analysis of commercial and mobile APIs that highlights the widespread issue of non-compliance with core REST principles (including misuse of HTTP methods) and correlates non-adherence with negative impacts on maintainability and development. This is confirmed by Delphi studies (e.g., Kotstein and Bogner [30]). Ehsan et al. [12] performed a systematic literature review on REST sevices and discuss how non-standard practices (which include HTTP method misuse) contribute to increase testing complexity. They point out that automated tools rely on the predictable semantics of the HTTP methods to structure test generation and validation, making non-compliant APIs difficult to test effectively. Palma et al. [41] highlighted this anti-pattern already in 2015, while more recent works confirm that such a problem is still present today in web APIs [6, 42], IoT applications [43], and microservices [11]. For instance, Dey et al [11] reports that 8% of the considered in production API operations exhibit the CRUD anti-pattern. Such studies highlight that this issue also affects flagship products like Google Cloud, IBM Watson IoT, Microsoft Azure, Dropbox, and Amazon AWS IoT Core. Fixing this problem with manual analysis of the codebase is not a viable option since the architecture of REST APIs is typically complex or not fully available for inspection. Hence, it becomes extremely important to have automated support for understanding the actual CRUD semantics of REST API operations.

In this paper, we propose an automated approach to infer the CRUD semantics of REST API operations. The approach aligns with the most common practice of REST API testing tools [19] by only relying on HTTP interactions with the API i.e., in a black-box fashion. The inference process works incrementally by refining the knowledge about operation CRUD semantics after each interaction with the API. Specifically, the approach starts from tentative semantics for the operations (e.g., retrieved by the corresponding HTTP methods). It then crafts test cases for the operations (i.e., HTTP interactions) derived from specific *CRUD test patterns*. The latter are interaction patterns meant to confirm operation semantics distilled from the intended semantics of each CRUD verb. For instance, to confirm that the CRUD semantics of an operation is *Create*, one can invoke the operation to supposedly create a resource, and then check if a fresh resource has been indeed created (by leveraging a read operation on the same resource, thus yielding a *read-after-create* testing pattern). The outcome of the test is used to *confirm* the semantics of some operations (the inferred semantics may not necessarily coincide with the initial one). Of course, the effectiveness of such test cases depends on how much the CRUD semantics knowledge adheres to the actual implementation. This is why the information about confirmed operations, i.e., operations having their CRUD semantics confirmed, is used to craft new testing scenarios for other operations, within an iterative process. In the beginning, test cases are likely to confirm a few operations, while iteration after iteration, the approach is supposed to confirm more and more operations by refining their CRUD semantics. When all operations have been confirmed, or when the inference budget expires, the process halts, yielding the inferred CRUD semantics for the confirmed operations. It may happen that some operations remain unconfirmed. In such cases, the approach signals the inability to infer operations' CRUD semantics. We implemented the approach into a tool named CRUDinfer, and empirically evaluated its performance on benchmark REST APIs. The results highlight the remarkable accuracy of the approach. Indeed, CRUDinfer achieved an overall precision of 95% when inferring operation CRUD semantics for the considered API operations without any prior CRUD semantics knowledge. The evaluation also highlights the refinement capabilities of the approach, which improves the precision of operation CRUD semantics statically retrieved from the OpenAPI specification by baseline inference approaches. Specifically, CRUDinfer improved, overall, by +38.8% and +15.1% the precision of inference approaches based on Natural Language Processing (NLP) techniques and Large Language Models (LLMs), respectively.

Paper Structure. Section 2 provides the background about REST APIs and OpenAPI specifications. Then, the details of the proposed

```yaml
paths:
  /users:
    get:
      operationId: get_users
      responses:
        '200':
          description: See all details of the users
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/User"
  /users/register:
    post:
      operationId: register_new_user
      requestBody:
        content:
          application/json:
            required: true
            schema:
              $ref: "#/components/schemas/User"
  /users/{username}:
    delete:
      operationId: delete_user
      parameters:
        - name: username
          in: path
          required: true
          schema:
            type: string
components:
  schemas:
    User:
      type: object
      properties:
        username:
          type: string
        password:
          type: string
```

Figure 1: Part of the OpenAPI specification for *VAmPI*.

inference approach are described in Section 3. In the subsequent Section 4, the approach is empirically evaluated on benchmark case studies. The related work is reported in Section 5. Finally, Section 6 concludes the paper and discusses possible future work.

2 Background

2.1 RESTful APIs

A RESTful (or REST) API is a web API that adheres to the REpresentational State Transfer architectural style [14]. REST APIs provide a uniform interface to create (*C*), read (*R*), update (*U*), and delete (*D*) resources (known as CRUD semantics). A resource is identified by a HTTP URI, and CRUD operations are typically mapped to the HTTP methods `POST`, `GET`, `PUT` (or `PATCH`) and `DELETE` [15]. As an example, we consider *VAmPI* [13], an open source REST API to manage users and books of a library. The HTTP URI pointing to a user resource is `/users`. In this case, the operation `GET /users` is used to retrieve the list of registered users, and the operation `POST /users/register` to register a new user in the system. The REST API may accept input parameters to specify additional information for executing an operation, such as the identifier of the user to delete (e.g., in `/users/{username}`) or a structured object to be registered in the system in the body of the request with the `POST` method.

2.2 The OpenAPI Specification

OpenAPI [24] defines a standard to document REST APIs. According to it, a web service is described using a structured file (YAML or JSON) that specifies how to reach the service using a URI, which authentication schema is adopted, and the details of all the operations available in the API. In particular, for each operation the input parameters (and their schema) to be used in requests, and the schema of responses are given. Figure 1 contains an excerpt of the OpenAPI specification for *VAmPI*. After an initial header that specifies versions, licenses, and the base URL of the API (not shown in the code for space reasons), the OpenAPI specification contains an array of *paths*, namely the list of URL paths available in the API. In our example, we focus on two URL paths: `/users`, and `/users/register`. Each path supports one or more HTTP methods. A path together with an HTTP method composes an *operation*, which is usually identified by an *operation ID*. For instance, the method `GET` in `/users` (*get_users*) is used to retrieve the list of all registered users, while the method `POST` in `/users/register` refers to the operation *register_new_user*, meant to register a new user.

Input and output of operations are associated to a *schema* that specifies their type and, optionally, a set of constraints on values (e.g., a *minimum* or *maximum* value for numeric parameters). Types can be atomic (e.g., integers and strings) or structured (i.e., compound objects). For instance, the *register_new_user* operation expects a request body containing a JSON object with three string fields: `username`, `password`, and `email`. Such fields are encapsulated into a `User` object, whose schema is provided in the *components* section of the specification. Similarly, the response schema of the *get_users* operation is a JSON array of `User` objects.

To provide inputs when exercising an operation, *path parameters* can be used. An example is the operation *delete_user*, where the user to delete is identified by the field `username` provided to the API via the path parameter `/users/{username}`.

3 Refinement-based CRUD Inference

3.1 Approach Overview

To infer the CRUD semantics of REST API operations, we propose CRUDinfer, an automated approach that builds knowledge about operations semantics simply by interacting with the target API via the HTTP interface. This knowledge is iteratively refined by observing the outcome of the interactions with the API, yielded after specific test scenarios. The overview of the approach is depicted in Figure 2. CRUDinfer takes as input the OpenAPI specification of the target API and collects all available operations. It then hypothesizes an initial CRUD semantics for such operations, which will be refined during the subsequent inference process. The initial semantics can be user-provided (e.g., retrieved with a third-party tool) or empty. In the latter case, CRUDinfer assumes as initial semantics the HTTP methods found in the specification.

The CRUD semantics of all operations is initially considered unconfirmed. Then, the inference process starts by picking an unconfirmed operation with the aim of checking its actual, hypothesized CRUD semantics or refining it in case it detects incorrect semantics. The confirmation/refinement relies on the outcome of specific test cases, concretized from specific *CRUD test patterns*. The idea is to exploit interaction patterns modeling intrinsic properties derived from the CRUD discipline in order to craft test cases trying to confirm the actual CRUD semantics of an operation. The result of such test cases is checked against specific *CRUD oracles*: in case of success the actual CRUD semantics is confirmed and the operation is removed from the unconfirmed operations pool.

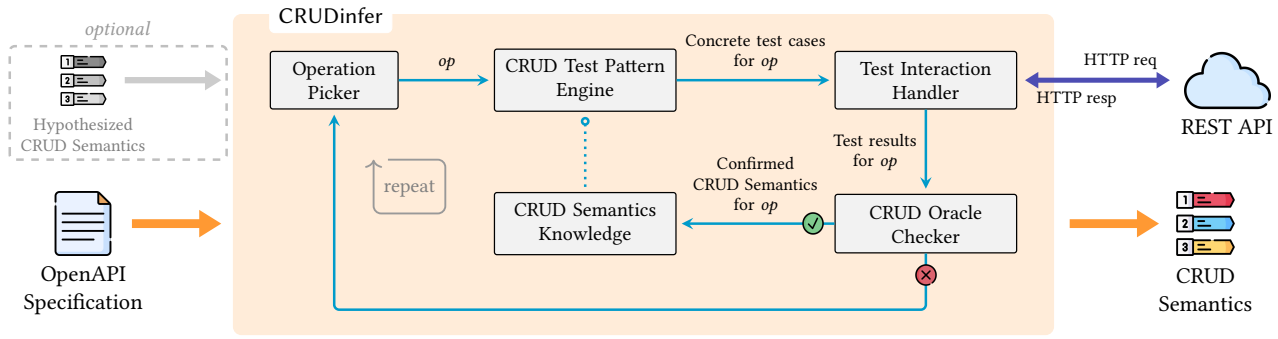

Figure 2: Overview of CRUDinfer Approach.

Crafting effective test cases depends on the actual (global) knowledge about operations CRUD semantics. As an example, to confirm an operation as a *Create*, we can craft a sequence of API calls that first attempts to create a resource through the operation under confirmation and then checks if the resource has been indeed created by leveraging a *Read* operation on the same resource. The prerequisite is to have already confirmed the semantics of a *Read* operation. If this does not apply, the confirmation of the operation as a *Create* is postponed, after refining the knowledge about other operations. This is why the inference process is iterative. In the beginning, since the reliability of operations CRUD semantics is probably low, the inference process is likely to confirm the semantics of a few operations. However, iteration after iteration, the approach is expected to confirm more and more operations, by refining their CRUD semantics.

Operations are selected for confirmation with different priority to minimize the probability of failing interactions. For instance, operations that do not require inputs are selected first because they are more likely to succeed. Similarly, *Read* operations should be confirmed earlier than every other type of operation since, as we will see in the next subsections, they may provide data needed to exercise *Create*, *Update*, or *Delete* operations. We rely on *data dependencies* between operations, that is, producer-consumer relations between operation parameters, to set selection priorities.

When test cases fail, namely the CRUD oracles do not confirm the hypothesized semantics of an operation, the approach tries to infer an alternative operation semantics. It first tries to guess the correct semantics by inspecting the (failing) test cases executed so far. For instance, if the confirmation of a hypothesized *Update* operation on resources of type τ fails, by inspecting the executed test cases the approach may guess that the operation is a *Create* operation (e.g., by realizing that a new instance of a τ resource has been added to the API). The approach then tries to confirm the guessed semantics. In case of another failure, the approach tries to infer an alternative operation semantics by applying all CRUD test patterns. The semantics associated with the first pattern yielding to successful test cases is assigned to the operation under confirmation. When even such a strategy is not able to infer a CRUD semantics, the operation confirmation is postponed. Indeed, the inability to confirm a CRUD semantics may be due to a limitation of test case generation (e.g., incorrect parameter value generation). After confirming the semantics of other operations and collecting more testing data, the confirmation of the operation semantics can be retried, with a higher chance of yielding a positive outcome.

When the inference budget expires, the CRUD semantics of the confirmed operations is output by CRUDinfer. Specifically, the OpenAPI specification is annotated with the inferred operation CRUD semantics, and the tool warns about unconfirmed operations.

Usage Scenarios. CRUDinfer is essentially a tool that automatically finds inconsistencies between the documentation (in the form of an OpenAPI specification) of a REST API and its implementation, providing a fix in most cases. Thus, the tool can be primarily used by developers to improve the quality of the API documentation (when the mismatch is due to a flaw in the OpenAPI specification) or identify misuses of HTTP methods (when the mismatch is due to a wrong implementation). Secondly, CRUDinfer can be used by researchers and developers as a support for REST API testing tools, by providing them with more accurate operations' CRUD semantics. Indeed, even if, in principle, CRUD-unaware testing tools would sooner or later probably be able to test any operation after enough repeated attempts, this can be very costly. Incorrect CRUD semantics would make tools waste valuable testing budget in repeated unsuccessful attempts, causing testing efficiency to drop, potentially to unbearable levels (especially in complex APIs). A challenging example is a *Delete* operation that uses the `GET` method (a common anti-pattern). Tools would fuzz it intensively, causing a decrease in resource count. Thus, other operations could be attempted fewer times, requiring a longer time to reach adequate coverage and harming testing efficiency and testing cost.

In the rest of the section, we will first present the CRUD test patterns we designed (in Subsection 3.2), together with the details of the test case generation process built upon these patterns (in Subsection 3.3). Then, we will provide the CRUD oracles adopted to confirm the operation semantics by inspection of the test results (in Subsection 3.4). An example of the inference process is provided at the end of the section (in Subsection 3.5).

3.2 CRUD Test Patterns

In order to craft test cases confirming API operations CRUD semantics, we defined a catalog of CRUD abstract test patterns by using the pattern syntax proposed by Corradini et al. [9]. These patterns represent abstract interactions with the API that, once instantiated, will result in actual sequences of HTTP requests. Specifically, a *CRUD test pattern* consists in a non-empty sequence of CRUD

labels, which can be: C_τ (*Create* a resource of type τ); R_τ (*Read* a resource of type τ); RM_τ (*Read* an array of resources of type τ); U_τ (*Update* a resource of type τ); and D_τ (*Delete* a resource of type τ). For instance, $C_{\texttt{User}}$ represents an operation, among those available in the API specification, that creates resources of type `User`; while $RM_{\texttt{User}}$ represents an operation that retrieves a list of resources of type `User`. When defining CRUD test patterns, we identify subsequences by enclosing a sequence of labels within curly brackets, and we denote with a question mark the fact that a sequence is optional. In a CRUD test pattern, operations are normally supposed to succeed (i.e., the API should reply with a `2XX` response code). When an operation is intentionally deemed to fail (i.e., the API replies with a `4XX` or a `5XX`), we attach the corresponding label in the pattern with the $^\times$ symbol. Finally, unconfirmed operations in a CRUD test pattern are overlined. For instance, the pattern $\langle \overline{C}_\tau, C_\tau, U_\tau^\times, (R_\tau)^? \rangle$ can be instantiated with: two *Create* operations for τ resources, the first picked from the pool of unconfirmed operations and the second from the confirmed one; an *Update* operation for τ resources supposed to fail; and an optional *Read* operation for τ resources.

Each CRUD test pattern defines the kind and order of operations to be used to generate concrete test cases, in order to confirm the hypothesized semantics for a given operation. When analyzing the result of the execution of test cases derived from CRUD test patterns, we will able to witness that a *Create* operation creates resources, a *Read* operation reads resources, a *Delete* operation deletes resources, and an *Update* operation modifies resources.

Read Pattern. When reading resources of type τ, the expected effect of a *Read* operation is to retrieve a τ resource, or a list of τ resources, without performing any modification on τ resources, including those retrieved. Hence, to confirm an operation $\overline{R}_\tau$ (the same applies to $\overline{RM}_\tau$) we will use the pattern:

$$\langle (RM_\tau)^?, \overline{R}_\tau, \overline{R}_\tau, (RM_\tau)^? \rangle \tag{1}$$

By comparing the outcome (zero, one, or multiple τ resources) of two subsequent readings, we can check if the operation produces side effects on τ resources. Moreover, by comparing all τ resources before and after the execution of $\overline{R}_\tau$ (by means of the enclosing, confirmed, RM_τ operations) we will be able to notice if the number of τ resources has been altered (it should not). The enclosing RM_τ operations are optional since they may not be available in the API. Note that, the pattern can be directly applied when the *Read* operation to confirm does not require inputs (e.g., an operation retrieving all users). Instead, if the operation requires input data (e.g., an operation retrieving a user with a specific username), some preliminary API calls must be performed to gather valid input data from API responses. This point is explained in detail in the next subsection (Subsec. 3.3, *Sequence Setup* paragraph).

Create Pattern. When creating a resource of type τ, the expected effect of a *Create* operation is to add the new resource among the existing ones, incrementing the number of resources of type τ. Hence, to confirm an operation $\overline{C}_\tau$ we will use the pattern:

$$\langle (RM_\tau)^?, \overline{C}_\tau, R_\tau, (RM_\tau)^? \rangle \tag{2}$$

The *Read* operation R_τ is used to check if the content of the added resource corresponds to one passed to $\overline{C}_\tau$ (consistency check). Similarly to the *Read Pattern* case, by comparing all τ resources before and after the execution of $\overline{C}_\tau$ we will be able to notice if a new resource has been added. Again, the enclosing RM_τ operations are optional since they may not be available in the API. In that case, we can check if a resource has been correctly created but not the number of τ resources before and after creation.

Update Pattern. When updating a resource of type τ, the expected effect of an *Update* operation is to modify the attributes of a resource without altering other τ resources. Hence, to confirm an operation $\overline{U}_\tau$ we will use the patterns:

$$\langle RM_\tau, (C_\tau)^?, \overline{U}_\tau, R_\tau, RM_\tau \rangle \qquad \langle C_\tau, \overline{U}_\tau, R_\tau \rangle \tag{3}$$

The first is used when the API provides an operation RM_τ to retrieve a list of τ resources, while the second when the API does not. In the latter case, we assume the availability of an operation C_τ creating τ resources (it is unlikely that an API does not provide RM_τ or C_τ operations, meaning that it is not possible to operate on τ resources).

In both patterns, the *Create* operation C_τ is meant to create a fresh τ resource, which will be fed to $\overline{U}_\tau$. In the first pattern, C_τ is optional since the API may not provide an operation to create τ resources, or the approach has not yet confirmed one yet. In that case, we will rely on a resource retrieved by the initial RM_τ. By comparing all τ resources before and after the execution of $\overline{U}_\tau$ we will be able to notice if the number of resources has been altered. When no RM_τ operation is available (second pattern), we cannot check that the number of τ resources before and after the update is unchanged. In both patterns, the *Read* operation R_τ is used as a consistency check, verifying the propagation of the modified values.

Delete Pattern. When deleting a resource of type τ, the expected effect of a *Delete* operation is to remove the resource from the existing ones, decrementing the number of resources of type τ. Additionally, a resource cannot be removed twice. Hence, to confirm an operation $\overline{D}_\tau$ we will use the patterns:

$$\langle RM_\tau, (C_\tau)^?, \overline{D}_\tau, \overline{D}_\tau^\times, RM_\tau \rangle \qquad \langle C_\tau, \overline{D}_\tau, \overline{D}_\tau^\times \rangle \tag{4}$$

Similarly to the *Update Pattern* case, the first pattern is used when the API provides an operation RM_τ to retrieve the list of τ resources, while the second when the API does not. Again, in the latter case, we assume the availability of an operation C_τ creating τ resources.

In both patterns, the *Create* operation C_τ is meant to create a fresh τ resource, which will be fed to $\overline{D}_\tau$. In the first one, C_τ is optional since the API may not provide an operation to create τ resources (or the approach has not yet confirmed one). In that case, we will rely on a resource retrieved by the initial RM_τ. By comparing all τ resources before and after the execution of D_τ we will be able to notice if the correct resource has been removed. When no RM_τ operation is available (second pattern), we cannot check that the number of τ resources before and after the update is decreased. In both patterns, we execute $\overline{D}_\tau$ twice to ensure a resource cannot be removed multiple times. So, the second deletion is supposed to fail.

3.3 Concrete Test Case Generation

CRUD test patterns are used to verify the hypothesized semantics for a given operation, by constructing concrete test cases, namely *concrete sequence* of HTTP requests, for the API. Specifically, for each label of the pattern a corresponding operation from the REST

API is chosen among those that have already been confirmed, and an HTTP request for each operation is crafted. The selected operations for a pattern must work on the same instance of the τ resource of the operation to confirm. As an example, in the *Update* pattern (3) the $\overline{U}_\tau$ operation is fed with the same τ resource instance previously created with the C_τ operation. A concrete sequence is considered successfully constructed if, for each label required by the pattern, it contains a corresponding operation of the REST API. Since we may have multiple candidate operations with the same label, multiple different concrete sequences might be available for instantiation. Nevertheless, when the pattern contains multiple instances of the same label, the same concrete operation will be selected to replace the label. For each operation of the concrete sequence, an HTTP request is crafted, by populating all its input values when needed.

Sequence Setup. Some operations require no input in order to be executed (e.g., an operation retrieving all users), while others need to fetch some data (e.g., an operation retrieving a user with a specific username). In the latter case, input data is selected with the following strategy. We first try to randomly craft input values, which are supposed to succeed when the operation parameter is reasonably easy to guess (e.g., the age of a person). In case of failures, we resort to response data of other operations to use as input values for parameters hard to guess (e.g., the unique id of a resource). Hence, a test case is possibly prefixed with a sequence of "setup" operations that retrieve the data needed to exercise the operation under confirmation. For instance, to confirm an operation that selects users by id, we may setup the test case by executing an operation retrieving all user ids. The setup sequence may contain multiple operations, in the number sufficient to fetch all data needed by the operation under confirmation. To improve efficiency, all data fetched during test case generation is saved into a dictionary and reused in subsequent test cases when possible. This decreases the number of setup interactions with the API needed. Note that, *Read* operations, which usually provide many example values, are selected for confirmation with high priority. Hence, in practice, when considering operations requiring some inputs the approach is likely to have many values in the dictionary to try.

Resource Identifier. As already discussed, operations belonging to the same CRUD test pattern should operate on the same resource instance. This requires to infer how a resource is identified by the API. We adopt the heuristic proposed by Corradini et al. [9], based on typical practices of REST API development. Specifically, we consider the parameter ending with the suffix "id" or "name" as a resource identifier. When multiple parameters align with this heuristic, we prioritize them in the following order: *1)* parameters in path variables; *2)* query parameters; *3)* parameters in the request body, beginning with the outermost level of nested objects inward. Once the identifier of a τ resource is inferred, the same identifier is used in all operations of a test case working on τ resources, thus guaranteeing to operate on the same resource instance.

3.4 CRUD Oracles

Once concrete test cases are executed, CRUD oracles evaluate if the behavior of the operation under confirmation conforms to the hypothesized CRUD semantics. CRUD oracles rely on the list of HTTP interactions recorded for a test case, one request and one response for each operation in the sequence. Operation invocations are generally meant to succeed, hence, for each operation of a test case the oracle checks if the corresponding response is a success (`2XX` HTTP response). When operations are intentionally deemed to fail, the oracle checks if the corresponding response is a failure, either client-side (`4XX` HTTP response) or server-side (`5XX` HTTP response). The oracle also checks the constraints imposed by the CRUD test patterns as described in Subsection 3.2.

For instance, in a test case instantiated from the *Create* pattern $\langle (RM_\tau)^?, \overline{C}_\tau, R_\tau, (RM_\tau)^? \rangle$, the oracle considers the test case as passed when: *(i)* the HTTP request yielding from $\overline{C}_\tau$ returns a `2XX`; *(ii)* the content of the τ resource returned by the HTTP request yielding from R_τ matches with the one used in the request yielding from $\overline{C}_\tau$; and *(iii)* the responses of the HTTP requests yielding from the first and the second RM_τ return two lists differing by one element (one more element must be present in the second response).

In a test case distilled from the *Delete* pattern $\langle C_\tau, \overline{D}_\tau, \overline{D}_\tau^\times \rangle$, instead, the oracle considers the test case as passed when: *(i)* the HTTP requests yielding from C_τ and $\overline{D}_\tau$ both return a `2XX`; and *(ii)* the HTTP request yielding from $\overline{D}_\tau^\times$ returns a `4XX` or a `5XX`.

When the oracle tags a test case as passed, the corresponding operation is confirmed with the semantics of the CRUD test pattern from which the test case has been concretized. In case of failures, a test case is retried (up to a fixed number of attempts) with different values for the input data for the operations in the sequence. Indeed, it may be the case that the semantics of an operation under confirmation is correct, but the data needed to make its execution successful has not been correctly crafted (e.g., the adoption of a name for a user not yet inserted in the API's users database).

3.5 Running Example

Suppose that during the inference process for the *VAmPI* API introduced in Section 2 we aim to confirm the *Delete* semantics for the operation `delete_user`, that works on `User` resources. Suppose that the *Read* semantics for the operation `get_users`, working on lists of `User` resources, has been already confirmed, while the *Create* semantics for the operation `register_new_user`, working on `User` resources, has not been confirmed yet. Then, the approach tries to apply the CRUD test pattern (4). Since `register_new_user` is unconfirmed, the approach does not have suitable candidates for instantiating the $C_{\texttt{User}}$ label of the pattern. Instead, `get_users`, being already confirmed, can be used to instantiate the $RM_{\texttt{User}}$ labels of the pattern. This yields the following concrete test case:

$$\langle \texttt{get\_users}, \texttt{delete\_user}, \texttt{delete\_user}, \texttt{get\_users} \rangle$$

where the second deletion is supposed to fail. In this particular case, no input is required to execute the operations in the sequence. Still, the sequence should operate on the same resource instance. Since `User` objects contain a field name `username`, the heuristic described in the previous subsection suggests selecting such a field as a resource identifier. Hence, after executing the first `get_users` request, the `username` of one of the `User` objects retrieved is feed to the subsequent `delete_user` operations.

Once the response to the requests yielding from the operations in the sequence are collected, the CRUD oracle is applied. Specifically, the oracle checks that the response to the first `delete_user`

Table 1: Benchmark REST APIs.

API	# Op	Source
Language Tool	2	Kim et al. [27]
Genome Nexus	23	Kim et al. [27]
REST Countries	22	Kim et al. [27]
Person Controller	12	Kim et al. [27]
Proj. Tracking System	59	Kim et al. [27]
Oshome API	122	Kim et al. [28]
Expense Tracker	12	GitHub [38]
REST API Demo	8	GitHub [7]
Movie Ticket Booking	22	GitHub [46]
RBAC User Mgmt.	15	GitHub [1]
Ecommerce MVC	41	GitHub [34]
OWASP crAPI	41	GitHub [16]
DVAPI	16	GitHub [45]

request is a `2XX`, while the response to the second `delete_user` is either a `4XX` or a `5XX`. Additionally, the oracle checks that the list of `User` resources retrieved by the second `get_users` contains one less element than the list of `User` resources retrived by the first `get_users`. If both checks are positive, then the *Delete* semantics for the `delete_user` operation is confirmed.

4 Experimental Validation

We empirically validated the capabilities of CRUDinfer in inferring operation CRUD semantics on a benchmark collection of REST APIs. The tool has been implemented by using the RestTestGen Framework [10], which allows the quick development of black-box REST API testing strategies. The framework offers a solid OpenAPI specification parser, an HTTP interactions handler, built-in value generators, and many ready-to-use testing components. The experiments have been guided by the following research questions.

RQ_1 How accurate is the proposed automated CRUD semantics inference approach?

RQ_2 Is the proposed inference approach helpful in improving the accuracy of hypothesized CRUD semantics provided by an external source?

With **RQ_1**, we aim to assess how accurate CRUDinfer is in inferring operation CRUD semantics when no hypothesized semantics is given. This represents an in-the-wild setting, where the tool is asked to automatically infer CRUD semantics without any prior knowledge about the REST API except its OpenAPI specification. With **RQ_2**, we aim to assess the refinement capabilities of CRUDinfer, measuring the accuracy improvement w.r.t. baseline approaches. This represents a controlled setting, where the tool is asked to infer CRUD semantics starting from a provided hypothesized one.

4.1 Benchmark APIs

We sourced six APIs from previous studies of Kim et al. [27, 28], and seven APIs from GitHub. We excluded the APIs from Kim et al. [27, 28] not interesting or not compatible with our evaluation infrastructure. Specifically, APIs such as *NCS*, *SCS*, and *REST Countries* have only `READ` operations, all using the correct CRUD verb. Validating these operations would have been easy for our approach, so not very interesting. We kept only one of them (*REST Countries*), randomly selected, to avoid inflating performance results. Other APIs, such as *Market*, require cookie-based authentication, not supported by the RestTestGen Framework we used for the implementation.

Table 2: CRUD semantics distribution in the benchmark.

CRUD Semantics	# Op	Wrong HTTP Method
`CREATE`	55 (13.9%)	1 (1.8%)
`READ`	183 (46.3%)	68 (37.2%)
`READ_MULTI`	83 (21.0%)	10 (12.0%)
`UPDATE`	38 (9.6%)	10 (26.3%)
`DELETE`	30 (7.6%)	1 (3.3%)
Not CRUD	6 (1.5%)	–

The additional seven APIs have been selected after a search on GitHub, taking the most starred results. Among such a list, we took the first seven we managed to make work. We made some little changes to the original versions of some APIs from GitHub. These changes are minimal and limited only to the correction of logical errors in the code. We also generated an OpenAPI specification when not provided. Specifications have been generated from source code by using the tool Respector [23].

The list of the thirteen APIs used in the evaluation, together with the number of exposed API operations, is reported in Table 1. These APIs represent a wide range of application domains and levels of complexity while ensuring comprehensive coverage of CRUD operation semantics (as we can see from Table 2). The size and complexity of the dataset align with those used in literature [8, 27].

Even if we selected case studies in the wild without specifically looking for CRUD semantics mismatches, among the 395 operations in the dataset, 90 of them (22.8%) use an incorrect HTTP method. Specifically, 37.2% of `READ`, 26.3% of `UPDATE`, 12.0% of `READ_MULTI`, 3.3% of `DELETE`, and 1.8% of `CREATE` are wrongly used. This aligns with the data from previous studies [6, 11, 42, 43], confirming that CRUD semantics mismatch is an actual issue in REST APIs.

4.2 Collected Metrics

To measure the performance of CRUDinfer we adopted standard information retrieval metrics.

Precision The ratio of the operations for which a CRUD semantics has been correctly inferred (true positives) to all operations for which such a CRUD semantics has been inferred (true and false positives), i.e., $\mathsf{Pr} = \mathsf{TP}/\mathsf{TP+FP}$.

Recall The ratio of the operations for which a CRUD semantics has been correctly inferred (true positives) to all operations actually having such a CRUD semantics (true positives and false negatives), i.e., $\mathsf{Re} = \mathsf{TP}/\mathsf{TP+FN}$. We consider the cases when the approach is not able to infer the CRUD semantics as false negatives.

Accuracy The ratio of the operations for which a CRUD semantics has been correctly inferred (true positives) to all inferences for such a CRUD semantics (true/false positives and false negatives), i.e., $\mathsf{Acc} = \mathsf{TP}/\mathsf{TP+FP+FN}$.

F-measure An aggregate metric combining Precision and Recall, i.e., $\mathsf{F1} = 2 \cdot (\mathsf{Pr}\cdot\mathsf{Re}/\mathsf{Pr+Re})$.

The ground truth has been manually retrieved by one of the authors and by an external collaborator expert in web development. The two annotators worked independently and made their decisions based on the inspection of the API source code. Disagreements have been solved with the help of a third expert acting as an arbiter.

4.3 Experimental Procedure

We executed CRUDinfer on each REST API in the benchmark. During test case generation, the tool instantiates CRUD test patterns to concrete HTTP request sequences. Such patterns may yield many different concrete sequences, depending on the operations available in the API. We then set a `max_gen_tests` parameter to 100 indicating the maximum number of concrete test cases to generate for each CRUD test pattern. Since the outcome of test cases may be influenced by input value generation, a concrete HTTP request sequence is retried with different values for operation inputs. We then set a `max_op_attempts` and a `max_seq_attempts` parameters to 4 and 7, respectively, limiting the number of retries for failing operations in a sequence and for failing sequences, respectively. We tuned such values on an API not in the benchmark in order to achieve a good trade-off between semantics inference accuracy and time. For each API, CRUDinfer has been executed ten times to account for non-deterministic behaviors (e.g., random input generation) and average results have been collected. Before each run of the tool, APIs have been set to the same initial conditions.

To answer $\mathbf{RQ_1}$, we provided CRUDinfer with the OpenAPI specification of the REST APIs under test only. To answer $\mathbf{RQ_2}$, we additionally provided the tool with hypothesized semantics retrieved from baseline approaches. A hypothesized semantics, as well as CRUDinfer output, consists of an OpenAPI specification enhanced with custom `x-crudOperationSemantics` tags documenting CRUD semantics for API operations. Such tags have been manually checked against the ground truth to compute accuracy metrics.

Since, to the best of our knowledge, there is no baseline (e.g., comparing with REST API testing tools is challenging, since they do not have oracles for CRUD verbs), we implemented two approaches based on information available in the API documentation to automatically infer CRUD semantics that a developer would be likely to adopt. Indeed, we did not compare with the tools from Palma et al. (SODA-R [39], DOLAR [41], SARA [40]) and Bogner et al. (RESTRuler [6]) for two reasons. First, they do not support all the CRUD semantics mismatches that we address in the paper. Even if the authors list many anti-patterns, only a quite limited subset is supported by their tools. For instance, RESTRuler only supports two rules about CRUD semantics that detect misuse of the `GET` and `POST` methods only. Second, when running RESTRuler on the APIs in the benchmark, we noted that the tool missed almost all mismatches between CRUD semantics and HTTP methods. Indeed, only one mismatch has been found by the tool (a `POST` used to delete users in *Ecommerce MVC*). Conversely, the tools SODA-R, DOLAR, and SARA are not available. We also did not consider REST API testing tools in the comparison, since they do not explicitly provide oracles for CRUD semantics. During the testing process, it could be the case that they may manage to fix CRUD inconsistencies, but if this happens, it is not possible to exploit such information without having access to the implementation details of the tool.

The baseline approaches we implemented are described below. We remark that with the second research question we not aim to compare inference accuracy but, rather, to measure the improvement of inference accuracy. Hence, we believe the baseline selection and the methodological approach adopted by the baseline do not impact the confidence in the empirical results.

> You are an AI assistant for REST API testing. The user will ask for the semantics of each operations of an API. An API operation is identified by an endpoint, a HTTP method, and an operationId.
>
> Choose between the semantics CREATE, DELETE, READ, UPDATE, and OTHER the one that is the most correct. Decide after analyzing the following properties of an operation: endpoint, HTTP method, request parameters, and response parameters, provided by a subsection of the OpenAPI specification, in JSON format, associated with the API.
>
> The response must be on the form of
> (HTTP method, endopoint): inferred semantics
>
> Remember, you must chose only one semantics. If you are undecided between two or more semantics, please chose only one of them, the one you think is the most correct.

Figure 3: LLM system prompt.

NLP-based Inference. Operation CRUD semantics is inferred by leveraging Natural Language Processing (NLP) techniques. Specifically, we adopted a *word2vec* model [36] pre-trained on the Google News corpus [21] (3 million 300-dimension English word vectors) to compute the similarity between operation information from the OpenAPI specification (such as, operation summary, operation description, operation name, and operation path parameters) and CRUD verbs (i.e., create, read, update, delete, and other). The verb with higher similarity has been set as operation semantics.

LLM-based Inference. Operation CRUD semantics is inferred by leveraging a Large Language Model (LLM). Specifically, we applied *prompt engineering*[17] to interact with the pre-trained GPT4All model [37]. The model is contextualized as an assistant for REST API testing and asked to infer the CRUD semantics for all operations of the API whose OpenAPI specification is provided as prompt input. The system prompt used is reported in Figure 3.

4.4 Empirical Results

Table 3 reports the inference accuracy of CRUDinfer when no hypothesized semantics is given. Each line reports the collected metrics (i.e., Precision, Recall, Accuracy, and F-measure) for the corresponding API. On the last line, overall results aggregating on operations from all APIs are reported. As we can see from the table, CRUDinfer achieves high precision in all case studies. Remarkably, the approach does not yield any false positive in most APIs. The case studies where CRUDinfer achieves lower precision are *Movie Ticket Booking* (0.84), *DVAPI* (0.75), and *OWASP crAPI* (0.62). False positives here were due to the wrong response status code returned by the APIs. For instance, in *Movie Ticket Booking*, delete operations return `2XX` even when deletion fails. Hence, when applying the *Delete* pattern (4), even if the second deletion is not performed internally by the API, our CRUD oracle is deceived into thinking that such deletion has been performed since it relies on the response status code (wrongly) returned by the API. This prevents CRUDinfer to confirm the `DELETE` semantics of those delete operations. Instead, such operations are assigned with the `READ` semantics in many cases since when all resources have been deleted the *Read* pattern (1) can be applied (the deletion is considered as a reading of empty arrays of resources). This yields some false positives.

Table 3: Inference accuracy results for CRUDinfer.

API	Pr	Re	Acc	F1
Language Tool	1.00	1.00	1.00	1.00
Genome Nexus	1.00	0.70	0.70	0.82
REST Countries	1.00	0.50	0.50	0.67
Person Controller	1.00	0.50	0.50	0.67
Proj. Tracking System	0.98	0.86	0.84	0.91
Expense Tracker	1.00	0.90	0.90	0.95
REST API Demo	1.00	1.00	1.00	1.00
Movie Ticket Booking	0.84	0.49	0.43	0.58
RBAC User Mgmt.	1.00	0.80	0.80	0.89
Ecommerce MVC	0.91	0.90	0.82	0.90
Ohsome API	1.00	0.62	0.62	0.77
OWASP crAPI	0.62	0.21	0.18	0.31
DAPI	0.75	0.46	0.40	0.57
Overall	**0.95**	**0.65**	**0.63**	**0.77**

Still, the average precision of CRUDinfer is high (0.95), as well as the other collected metrics, which are overall never below 0.63, thus indicating a good general accuracy of the approach.

> **Answer to RQ_1**: *Empirical results highlight a remarkable accuracy of the proposed operation CRUD semantics inference approach, which exhibits a very low rate of false positives. Indeed, the overall precision of CRUDinfer on the benchmark REST API operations accounts for 95%.*

Table 4 reports the improvement of the inference accuracy results for baseline approaches when CRUDinfer is employed (i.e., CRUDinfer is executed with a hypothesized semantics provided by baseline approaches). Each line of the table reports the difference in the collected metrics (i.e., Precision, Recall, Accuracy, and F-measure) for the corresponding API in the benchmark w.r.t. the two baseline approaches (NLP-based inference and LLM-based inference). On the last line, overall results aggregating on operations from all APIs are reported.

As we can see from Table 4, CRUDinfer significantly improves the inference capabilities of baseline approaches. The hypothesized operation CRUD semantics from the baseline approaches is refined by CRUDinfer, achieving a higher precision in almost all case studies. Remarkably, for some REST APIs (such as *Language Tool*, *RBAC User Mgmt.*, and *Ohsome API*), the improvement in precision is more than 100.0%. In a few case studies, CRUDinfer introduces some false positives, with a slight decrease in precision (from 12.0% to 16.0%). After a manual inspection of these test outcomes, we noticed that the increase in false positives was due to the fact that CRUDinfer does not support non-CRUD operation semantics. Indeed, *OWASP crAPI* and *DVAPI* are login-based services, and login operations fall outside the CRUD semantics. Indeed, non-CRUD operations were correctly marked as `OTHER` by baseline approaches. Instead, since CRUDinfer does not have test patterns for non-CRUD operations, it ended up inferring a CRUD semantics for some non-CRUD operations, resulting in additional false positives.

The table highlights that the general increase in precision comes with a slight decrease of recall. This is somehow expected, as the proposed approach is based on confirmation. Indeed, CRUDinfer aims *to confirm* a hypothesized operation CRUD semantics, trying to correct it *when wrong*. Hence, the increase in precision indicates that the tool is effective in correcting wrong operation CRUD semantics inferred from baseline approaches. Dually, the inability to confirm a hypothesized semantics from CRUDinfer results in a false negative, thus decreasing recall. Indeed, CRUDinfer prioritizes a low false positive rate, by design. When confirmation/correction fails, the tool conservatively does not output any CRUD semantics. This leads to a false negative, decreasing recall. We privileged precision since a false positive indicates an uncaught mismatch between an operation's intended CRUD semantics and its actual implementation. Still, aggregate metrics show an increase in accuracy w.r.t. the NLP-based inference approach (+4.6% in Accuracy and +2.8% in F-measure), while a decrease w.r.t. the LLM-based inference approach (-22.0% in Accuracy and -13.4% in F-measure).

> **Answer to RQ_2**: *Empirical results highlight a good capability of the approach in refining a hypothesized operation CRUD semantics. Indeed, precision after applying CRUDinfer to baseline approaches based on NLP and LLM respectively increases by 38.8% and 15.1%, overall.*

Looking at the results, we can note a high variance of Accuracy among the different APIs. The latter is given by the fact that the approach is testing-based, hence subject to the typical challenges of randomized testing, like non-determinism and difficulty in generating effective input values. For instance, we noted that it was difficult for CRUDinfer to craft a valid request body for some *Create* operations in *Movie Ticket Booking*. For the same API, CRUDinfer is not able to confirm some defective *Delete* operations that return no error when trying to delete a non-existent resource. This defect deceives our tool when it tries to delete a resource twice, expecting an error on the second attempt that will never occur. In such cases, the tool may not be able to confirm the *Delete* semantics, resulting in an Accuracy drop. Another point impacting CRUDinfer performance is the lack of non-CRUD semantics awareness. We are planning to refine the semantics tags by adding login and other operation semantics as future work, but, currently, this is not supported by CRUDinfer. Still, to be as fair as possible, we considered non-CRUD operations (e.g., login) in the evaluation, even if not supported by CRUDinfer, resulting in additional false negatives for our tool.

Performance and Scalability. Even if the goal of the second research question is not to compare CRUDinfer's accuracy w.r.t. baseline inference approaches, we briefly comment here on such a comparison for the sake of completeness. Table 5 reports the accuracy metrics (i.e., Precision, Recall, Accuracy, and F-measure) for the hypothesized semantics from baseline approaches (only the overall results aggregating on operations from all APIs are reported, for space reasons). The table also reports the difference w.r.t. CRUDinfer with empty initial knowledge. As we can see, the results align with those of Table 4, with CRUDinfer showing higher precision (+38.2% w.r.t. the NLP-based approach and +14.7% w.r.t. the LLM-based approach), but lower recall (-22.4% w.r.t. the NLP-based approach and -34.2% w.r.t. the LLM-based approach). Aggregate metrics show a better accuracy w.r.t. the NLP-based inference approach (+3.6% in Accuracy and +2.2% in F-measure), while a worse accuracy w.r.t. the LLM-based inference approach (-23.4% in Accuracy and -14.3% in F-measure). Again, this is expected, as CRUDinfer prioritizes a low false positive rate, by design.

Table 4: Inference accuracy improvement results wrt baseline approaches.

API	CRUDinfer over NLP				CRUDinfer over LLM			
	Pr	Re	Acc	F1	Pr	Re	Acc	F1
Language Tool	+100.0%	+0.0%	+100.0%	+50.0%	+100.0%	+0.0%	+100.0%	+50.0%
Genome Nexus	+91.7%	-30.4%	+33.4%	+19.7%	+64.3%	-30.4%	+14.3%	+8.4%
REST Countries	+0.0%	-45.0%	-45.0%	-30.0%	+0.0%	-50.0%	-50.0%	-33.3%
Person Controller	+0.0%	-58.3%	-58.3%	-41.2%	-2.0%	-58.9%	-59.2%	-42.1%
Proj. Tracking System	-2.0%	+33.8%	+31.6%	+17.1%	-2.0%	-13.8%	-15.2%	-8.3%
Expense Tracker	-1.0%	-9.2%	-10.0%	-5.3%	-1.0%	-9.2%	-10.0%	-5.3%
REST API Demo	+0.0%	+0.0%	+0.0%	+0.0%	+0.0%	+0.0%	+0.0%	+0.0%
Movie Ticket Booking	+19.6%	-44.0%	-31.8%	-22.1%	-5.7%	-34.4%	-37.3%	-23.6%
RBAC User Mgmt.	+130.8%	-14.3%	+100.0%	+55.6%	+25.9%	-14.3%	+9.1%	+5.0%
Ecommerce MVC	-0.2%	-8.2%	-7.9%	-4.3%	-6.0%	-10.8%	-15.6%	-8.5%
Ohsome API	+138.1%	-27.1%	+44.2%	+28.3%	+47.1%	-38.0%	-8.8%	-4.9%
OWASP crAPI	+14.0%	-66.2%	-57.8%	-48.4%	-16.7%	-77.4%	-73.5%	-61.7%
DAPI	+29.3%	-34.3%	-14.9%	-10.9%	-12.8%	-50.0%	-50.0%	-36.0%
Overall	**+38.8%**	**-21.8%**	**+4.6%**	**+2.8%**	**+15.1%**	**-33.2%**	**-22.0%**	**-13.4%**

About scalability, in the experiments, inference time was of the order of 5/6 minutes, on average. Still, the performance of REST testing tools is not typically measured in terms of running time, since the latter is deeply affected by HTTP overhead. A more suitable measure could be the number of requests generated. In CRUDinfer, generation can be tuned in terms of the number of pattern concretizations and interaction retries, as explained in Subsection 4.3. We set such thresholds to relatively small numbers, still achieving good precision results. Hence, we believe scalability is not an issue.

4.5 Threats to Validity

Threats to the *internal validity* of empirical results are due to the metrics adopted to answer the research questions and the implementation of the baseline approaches. To mitigate these threats, we adopted standard metrics from information retrieval, such as Precision, Recall, Accuracy, and F-measure. Furthermore, the NLP and LLM models used in the evaluation were pre-trained, limiting potential bias. We addressed potential threats due to the non-deterministic components of the tool by running the experiments ten times and computing the average results.

Another internal threat is the manual creation of the ground truth, as the individual interpretation of the API business logic by the annotators could have affected the results. We limited this problem by asking the annotators to work in isolation and to reach consensus on any discrepancies, thus reducing the risk of bias.

Threats to *external validity*, impacting the generalization of our findings, are due to case study selection. We mitigated these threats by considering REST APIs adopted in previous studies, in addition to new APIs publicly available on GitHub. The size and complexity of the resulting dataset align with those used in literature.

5 Related Work

CRUDinfer is the first approach that dynamically validates the CRUD semantics of REST API operations, recommending fixes for inconsistent usage of HTTP methods. The approach crafts black-box test cases for a REST API, derived from specific *CRUD test patterns* and the API documentation. The feedback from test cases is used to iteratively refine the knowledge about the semantics of the API operations, with the ultimate goal of confirming their adherence to the correct CRUD principles. Mismatches resulting from the analysis can be reported to developers to find misuses in either the API implementation or its documentation.

Research on API design quality primarily focuses on identifying best practices, common mistakes, and the impact of design choices on software quality attributes like usability and maintainability. For instance, Delphi studies (e.g., Kotstein and Bogner [30], Tran et al. [49]) investigate how industry experts rate the importance of various REST design rules. These studies highlight that correctly using HTTP methods for CRUD operations is critical for achieving high usability, maintainability, and compatibility, suggesting that CRUD semantics usage is a foundational quality factor. Other work in this domain (e.g., Palma et al. [41, 42], Rodriguez et al. [48]) investigates typical anti-patterns present in REST APIs and how common they are in real-world applications. Nevertheless, a little less attention is given to the development of automatic approaches to spot and fix these anti-patterns. Palma et al. proposed several tools (i.e., SODA-R [39], DOLAR [41], SARA [40]) to automatically detect structural and behavioral best practices violations in REST APIs, based on linguistic analysis and natural language processing on API operation URLs and parameter names. Similarly, RESTRuler from Bogner et al. [6] aims at the same objective by applying pattern-matching rules on OpenAPI specifications. Differently from CRUDinfer, these approaches are static and cannot exploit the feedback from API execution. Moreover, their focus is on supporting a broad range of anti-patterns, providing a lightweight analysis for each of them (for instance, RESTRuler only provides two rules about CRUD semantics, modeling the `GET` and `POST` method misuse only). Instead, CRUDinfer is specific for operation CRUD semantics misuse, thus providing a more in-deep and effective analysis.

Literature about REST API testing comprises many different approaches, mainly working in a black-box setting, with the objective of maximizing test coverage. Black-box test case generators adopt various strategies. RESTler [4] and RestTestGen [8] exploit data dependencies (i.e., producer-consumer relations between operation parameters) to prioritize API operations to test first and use complex value providers to generate valid inputs for operation parameters. QuickREST [25] and Schemathesis [22] leverage property-based testing to verify API behavior against the API documentation (OpenAPI/GraphQL). Morest [32] extends this by dynamically modeling API behavior with a Property Graph, enabling the generation of

Table 5: Overall Accuracy of the hypothesized semantics from baseline approaches and comparison with CRUDinfer with empty initial knowledge (Table 3).

	Pr	Re	Acc	F1
NLP baseline	0.69	0.84	0.61	0.76
CRUDinfer empty	*+38.2%*	*-22.4%*	*+3.6%*	*+2.2%*
LLM baseline	0.83	0.99	0.82	0.90
CRUDinfer empty	*+14.7%*	*-34.2%*	*-23.4%*	*-14.3%*

context-aware test sequences. Dredd [2] validates responses against expected response status code, header, and body. RESTest [33] tests an API leveraing inter-parameter dependencies, producing nominal and faulty test cases. bBOXRT [31] performs robustness testing by feeding the API under test with malformed input generated using a collection of mutation operators, and observing how the API handles such invalid data. More recent tools perform test case generation by exploiting combinatorial approaches, like RestCT [50], and reinforcement learning, like ARAT-RL [27] (Q-learning), DeepREST [8] (Deep learning), and AutoRestTest [29] (Multi-agent Q-learning). Among the white-box approaches to REST API testing, the only work is EvoMaster [3], which generates test cases by exploiting evolutionary algorithms. Collectively, these tools aim to explore the API's input space and validate its responses, ensuring adherence to its documentation and detecting potential faults.

To the best of our knowledge, no work in literature aims at inferring the CRUD semantics of REST API operations. From a conceptual point of view, the closest works to ours are by Kim et al. [26, 28], in which the authors leverage NLP techniques and LLMs to enhance OpenAPI specifications with information retrieved from the human-readable portions of the specification. This information, such as example values or inter-parameter dependencies, can then be used by REST API testing tools to improve their effectiveness. This is similar in spirit to our approach: we enhance OpenAPI specifications with operation CRUD semantics information to support REST API testing tools. Instead, from a methodological point of view, the closest work to ours is by Corradini et al. [9], in which the authors propose abstract testing patterns tailored to spot mass assignment vulnerabilities (also known as broken object property-level authorization) in REST APIs. We indeed took inspiration from that work to build our CRUD test patterns. Despite the common idea of "abstract testing pattern", the objectives of our approach and the one proposed in Corradini et al. [9] are different: we focus on inferring operation CRUD semantics while in Corradini et al. [9] the focus is on vulnerability detection. Hence, despite being inspired by that work, only the core idea of abstract test patterns and their syntax have been taken from it. The application context and all the development are different and novel with respect to the paper introducing such a core idea.

6 Conclusion

While operations in a REST API are supposed to specify HTTP methods that match their CRUD semantics, this best practice is often disappointed by developers. This is a known anti-pattern widely diffused in REST APIs, as highlighted by literature studies. Mismatches between HTTP methods and operation CRUD semantics cause maintenance and evolution problems to developers, and therein the applicability of automated test case generation tools.

We then proposed a novel method to automatically infer the CRUD semantics of REST API operations via black-box testing. The approach iteratively refines the operation CRUD semantics starting from a, possibly empty, hypothesized one. When no hypothesized semantics is given, the approach assumes as an operation CRUD semantics its corresponding HTTP method. The hypothesized semantics is confirmed by iteratively executing CRUD test scenarios drawn from interaction patterns typical of CRUD operations. The approach is black-box, thus only requiring a REST API formal documentation adhering to the OpenAPI standard.

Empirical results on benchmark REST APIs highlight the accuracy of the approach, which achieves an overall precision of 95% when no hypothesized semantics is given. The proposed approach is also shown to improve the accuracy of a provided operations CRUD semantics. Specifically, we noted a boost in precision of +38.8% and +15.1% when considering baseline inference approaches based on NLP and LLM, respectively. Our evaluation confirms what the literature [6, 11, 41–43] highlights, namely that real-world REST APIs often fall short in adhering to development best practices, such as following CRUD principles. Manually mitigating this issue is not a viable option due to the magnitude of the incorrect implementations. Thus, automated solutions are advocated. Actual approaches look at the problem *statically*, by exploiting syntactic information retrieved from the API documentation (OpenAPI specifications) only. Still, our evaluation highlights the need for a more *dynamic* approach that can also exploit semantic information from the API execution. As shown by empirical results, this can lead to finding more best practice violations and implementation inconsistencies.

Another observation we can make is that CRUD semantics are sometimes too restrictive, not being able to model non-resource-centric scenarios, like authentication operations. The latter technically violate the REST paradigm, but they are essential in modern-day web services. We can think of extending CRUD semantics by adding a tag to OpenAPI specifications that models CRUD and non-CRUD verbs (e.g., login), with a mapping to the most suitable HTTP method to use in the latter cases. This would allow a controlled adoption of non-CRUD semantics for REST API operations.

Thus, an interesting future line of research, could be to extend the proposed approach to infer additional "kinds of semantics" for REST APIs. For instance, we may distill interaction patterns for operations performing login/signup, together with the corresponding oracles. This would help during manual or automatic security audits of REST APIs. Another possible extension could be to add support for user-defined interaction patterns in order to infer operation semantics closer to the REST API business logic.

Data Availability

The material to replicate our experiments is available on Zenodo [5].

Acknowledgments

This project has received funding from the European Union's HorizonEurope research and innovation programme under grant agreement No.101070238. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] andreagiassi. 2024. *RBAC User Management API.* https://github.com/andreagiassi/microservice-rbac-user-management

[2] apiaryio. 2023. *Dredd.* https://github.com/apiaryio/dredd

[3] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 28, 1 (2019), 1–37.

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful REST API fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.

[5] Pasqua Michele, Corradini Davide, Perlotto Michele, and Ceccato Mariano. 2025. *Replication package for the paper "CRUDinfer: Automated CRUD Semantics Inference for REST APIs Through Black-box Testing".* https://doi.org/10.5281/zenodo.15017219

[6] Justus Bogner, Sebastian Kotstein, Daniel Abajirov, Timothy Ernst, and Manuel Merkel. 2024. RESTRuler: Towards Automatically Identifying Violations of RESTful Design Rules in Web APIs. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE Computer Society, Los Alamitos, CA, USA, 123–134. doi:10.1109/ICSA59870.2024.00020

[7] callicoder. 2024. *REST API Demo.* https://github.com/callicoder/spring-boot-postgresql-jpa-hibernate-rest-api-demo

[8] Davide Corradini, Zeno Montolli, Michele Pasqua, and Mariano Ceccato. 2024. 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*. 1383–1394.

[9] Davide Corradini, Michele Pasqua, and Mariano Ceccato. 2023. Automated Black-Box Testing of Mass Assignment Vulnerabilities in RESTful APIs. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE Press, 2553–2564. doi:10.1109/ICSE48619.2023.00213

[10] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2022. RestTestGen: An Extensible Framework for Automated Black-box Testing of RESTful APIs. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 504–508. doi:10.1109/ICSME55016.2022.00068

[11] Krishno Dey, Hung Cao, and Francis Palma. 2024. Assessing the Linguistic Design Quality of APIs of Distributed Systems and Microservices. In *2024 34th International Conference on Collaborative Advances in Software and COmputiNg (CASCON)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–10. doi:10.1109/CASCON62161.2024.10838121

[12] Adeel Ehsan, Mohammed Ahmad M. E. Abuhaliqa, Cagatay Catal, and Deepti Mishra. 2022. RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions. *Applied Sciences* 12, 9 (2022). doi:10.3390/app12094369

[13] erev0s. 2025. *VAmPI.* https://github.com/erev0s/VAmPI

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

[15] Roy T. Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231. doi:10.17487/RFC7231

[16] OWASP Foundation. 2025. *OWASP crAPI.* https://github.com/OWASP/crAPI

[17] Louie Giray. 2023. Prompt Engineering with ChatGPT: A Guide for Academic Writers. *Annals of Biomedical Engineering* 51 (2023), 2629–2633. doi:10.1007/s10439-023-03272-4

[18] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 27 (2023), 41 pages. doi:10.1145/3617175

[19] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 27 (Nov. 2023), 41 pages. doi:10.1145/3617175

[20] Google. 2025. *Cloud API Design Guide.* https://cloud.google.com/apis/design/resources

[21] Google. 2025. *Google News corpus.* https://github.com/mmihaltz/word2vec-GoogleNews-vectors

[22] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving Semantics-Aware Fuzzers from Web API Schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ACM, New York, NY, USA, 345–346. doi:10.1145/3510454.3528637

[23] Ruikai Huang, Manish Motwani, Idel Martinez, and Alessandro Orso. 2024. Generating REST API Specifications through Static Analysis. In *IEEE/ACM Int. Conference on Software Engineering (ICSE)*. 1311–1323. doi:10.1145/3597503.3639137

[24] OpenAPI Initiative. 2025. *OpenAPI Specifications.* https://spec.openapis.org/oas

[25] S. Karlsson, A. Causevic, and D. Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 131–141. doi:10.1109/ICST46399.2020.00023

[26] Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. 2023. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1232–1243.

[27] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2023. Adaptive REST API testing with reinforcement learning. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 446–458.

[28] Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro Orso. 2024. 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*. 37–41.

[29] Myeongsoo Kim, Tyler Stennett, Saurabh Sinha, and Alessandro Orso. 2024. A Multi-Agent Approach for REST API Testing with Semantic Graphs and LLM-Driven Inputs. *arXiv preprint arXiv:2411.07098* (2024).

[30] Sebastian Kotstein and Justus Bogner. 2021. Which RESTful API Design Rules Are Important and How Do They Improve Software Quality? A Delphi Study with Industry Experts. In *Service-Oriented Computing*, Johanna Barzen (Ed.). Springer International Publishing, Cham, 154–173.

[31] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A black box tool for robustness testing of REST services. *IEEE Access* 9 (2021), 24738–24754.

[32] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-based RESTful API testing with execution feedback. In *Proceedings of the 44th International Conference on Software Engineering*. 1406–1417.

[33] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12571)*, Eleanna Kafeza, Boualem Benatallah, Fabio Martinelli, Hakim Hacid, Athman Bouguettaya, and Hamid Motahari (Eds.). Springer, 459–475. doi:10.1007/978-3-030-65310-1_33

[34] melardev. 2024. *Ecommerce MVC API.* https://github.com/melardev/SBootApiEcomMVCHibernate

[35] Microsoft. 2025. *Azure REST API Design Guidelines.* https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design

[36] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[37] Nomic. 2025. *GPT4All model.* https://gpt4all.io/index.html?ref=lookaitools

[38] pairlearning. 2024. *Expense Tracker API.* https://github.com/pairlearning/expense-tracker-api

[39] Francis Palma, Johann Dubois, Naouel Moha, and Yann-Gaël Guéhéneuc. 2014. Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach. In *Service-Oriented Computing*, Xavier Franch, Aditya K. Ghose, Grace A. Lewis, and Sami Bhiri (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–244.

[40] Francis Palma, Javier Gonzalez-Huerta, Mohamed Founi, Naouel Moha, Guy Tremblay, and Yann-Gaël Guéhéneuc. 2017. Semantic Analysis of RESTful APIs for the Detection of Linguistic Patterns and Antipatterns. *Int. J. of Cooperative Information Systems* 26, 02 (2017), 1742001. doi:10.1142/S0218843017420011

[41] Francis Palma, Javier Gonzalez-Huerta, Naouel Moha, Yann-Gaël Guéhéneuc, and Guy Tremblay. 2015. Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti)Patterns. In *Service-Oriented Computing*, Alistair Barros, Daniela Grigori, Nanjangud C. Narendra, and Hoa K. Dam (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–187.

[42] Francis Palma, Tobias Olsson, Anna Wingkvist, Fredrik Ahlgren, and Daniel Toll. 2022. Investigating the Linguistic Design Quality of Public, Partner, and Private REST APIs. In *2022 IEEE International Conference on Services Computing (SCC)*. 20–30. doi:10.1109/SCC55611.2022.00017

[43] F. Palma, T. Olsson, A. Wingkvist, and J. González-Huerta. 2022. Assessing the linguistic quality of REST APIs for IoT applications. *Journal of Systems and Software* 191 (2022), 111369. doi:10.1016/j.jss.2022.111369

[44] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. 2008. Restful web services vs. "big" web services: making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web* (Beijing, China) *(WWW '08)*. Association for Computing Machinery, New York, NY, USA, 805–814. doi:10.1145/1367497.1367606

[45] payatu. 2025. *Damn Vulnerable API.* https://github.com/payatu/DVAPI

[46] PraveenGirishNadumani. 2024. *MovieTicketBooking API.* https://github.com/PraveenGirishNadumani/SpringBoot_MovieTicketBooking

[47] RapidAPI. 2025. *API search engine.* https://rapidapi.com/search

[48] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. 2016. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In *Web Engineering (ICWE '16)*, Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso (Eds.). Springer International Publishing, Cham, 21–39.

[49] Van Tuan Tran, Manel Abdellatif, and Yann-Gaël Guéhéneuc. 2021. Formalising Solutions to REST API Practices as Design (Anti)Patterns. In *Service-Oriented Computing*, Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik (Eds.). Springer International Publishing, Cham, 153–170.

[50] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *Proceedings of the 44th Int. Conference on Software Engineering*. ACM, New York, NY, USA, 426–437. doi:10.1145/3510003.3510151