Combining Static and Dynamic Approaches for Mining and Testing Constraints for RESTful API Testing

Anonymous Author(s)

Abstract

In API testing, deriving logical constraints on API response bodies is crucial in generating the test cases to cover various aspects of RESTful APIs. However, existing approaches are limited to dynamic analysis in which constraints are extracted from the execution of APIs as part of the system under test. The key limitation of such a dynamic approach is its under-estimation in which inputs in API executions are not sufficiently diverse to uncover actual constraints on API response bodies. In this paper, we propose a static analysis approach in which the constraints for API response bodies are mined from API specifications. This static approach detects constraints uncovered by dynamic approaches, which rely on data obtained from API executions. We leverage large language models (LLMs) to comprehend textual descriptions in API specifications, mine constraints for response bodies, and generate test cases. To reduce LLM's hallucination and error propagation, we apply an Observation-Confirmation (OC) scheme which uses initial prompts to contextualize constraints, allowing subsequent prompts to more accurately confirm their presence. Our empirical results show that LLMs with OC prompting achieve high precision in constraint mining with the average of 91.2%. When combining static and dynamic analysis, our tool, RBCTest, achieves a precision of 78.5%. We also use its generated test cases to detect 21 mismatches in real-world APIs between their API specifications and actual response data. Four of those mismatches were, in fact, reported in developers' forums.

1 Introduction

By adhering to the principles of Representational State Transfer (REST), the RESTful APIs provide a standardized way for interoperability among components and software systems. RESTful API testing helps identify and resolve several issues, ensuring that APIs perform as expected [8, 10, 14, 24, 27, 33]. It also helps verify that APIs adhere to specifications and handle edge cases gracefully. Among techniques for API testing, black-box testing uses the OpenAPI Specification (OAS) as a basis to generate test cases and data [18, 22, 27]. The state-of-the-art API testing approaches are focused on status code [5, 6, 22, 33] and schema validation [5, 6, 33], even with rule extraction using human-readable descriptions in the OAS [19]. In *status code validation*, each HTTP request returns a response with a status code, a three-digit integer, indicating the outcome of the HTTP request. Current testing approaches define an oracle for a test case by validating whether the response status code matches the expected value. In contrast, *schema validation* ensures the correctness of the response data by checking it against a specified schema. This involves verifying the presence of all required properties and ensuring the data types of these properties match their schema in the specification.

While status code and schema validation effectively cover aspects of data representation and status checking, they may overlook the *logical correctness and validity of the response data from the APIs*, which is essential for software reliability. For example, an API request for a customer older than 18 receiving a response for one younger than 18 would not be detected by just validating the status code or schema. Deriving logical constraints on API response bodies is essential for generating test cases to cover RESTful APIs.

The state-of-the-art approaches for mining constraints on API response bodies focus only on *dynamic analysis* in which the constraints are extracted from the execution data of the system under test (SUT). AGORA [9] automatically detects *invariants*—properties of the output that should consistently hold true. To identify invariants, which serve as logical constraints on API response bodies, it extends Daikon [16], a dynamic instrumenter used to detect invariants during execution. As with dynamic analysis, it under-estimates the constraints due to the lack of diverse inputs to cover different logical aspects of API response bodies. For example, the inputs of the APIs might not be diverse enough to discover that the minimum age for an operation on a website is 18. Another important limitation of this approach is that it requires the SUT to operate accurately to extract the constraints from accurate inputs and outputs.

We propose RBCTest in which the dynamic analysis approach (when the execution data is available) is complemented with a LLM-based static analysis approach to mine the constraints of API response bodies from the API specification (when it is available). *Our static approach and AGORA complement to each other in the following ways.* First, constraint mining remains feasible even if only the API specification or execution data is available. For instance, when an API specification exists but the APIs and the system utilizing them are still under testing and development and may not function correctly, execution data is unavailable. Conversely, in a regression testing scenario with no up-to-date specification, the SUT with APIs from previous versions has been functioning, while the current version is still under development. In such cases, execution data from the previous version can be used to extract constraints to generate test cases for regression testing of the current version. Second, the specification often provides API details, including request bodies sent to the API and response bodies returned for each operation. This enables mining constraints on the API's response bodies to uncover more comprehensive information about these constraints, which might be overlooked by the dynamic approach. In contrast, due to actual execution, the runtime data helps derive more detailed constraints that are not defined in the specifications.

To mine constraints, we leverage the ability of large language models (LLMs) to comprehend natural language descriptions found in API specifications. Constraints on API response bodies are inferred from different sources, including response properties, response schema, operations, and request parameters. We also harness LLMs' proficiency in generating source code to create automated test cases from the mined constraints to verify if the SUT

correctly returns the content satisfying the mined constraints. Furthermore, we apply an *Observation-Confirmation* scheme. We divide the task of mining constraints into two phases: observation and confirmation. The initial prompt contextualizes the description of constraints, enabling the next prompt to more accurately decide their presence. As another issue in LLMs' exploration capability, they could produce resulting constraints that are not true. Thus, to improve the precision of the resulting constraints, we additionally enhance RBCTest with two extra mechanisms. First, before requesting the LLM to make observations concerning constraints on parameters, we perform *a filtering process* to keep only the valid ones. Second, after generating test cases for mined constraints, we add a *semantic verifier* to verify those test cases against the examples specified in the OAS file. The idea is that such examples tend to be correct because they illustrate the descriptions on the data types or the data specified in the OAS. For example, the OAS could give "March" as a valid month. If such a correct example does not pass a test case generated by RBCTest based on the mined constraint(s), the test case must be incorrect, which is caused by incorrect constraint(s). Thus, our verifier will discard them, leading to an improved precision.

We conducted several experiments to evaluate RBCTest using two datasets, one from the baseline [9] (AGORA dataset) with 11 operations in 7 API services and our RBCTest dataset collected from 8 real-world API services consisting of 59 endpoints and 83 operations [7]. Our empirical results show that RBCTest with GPT-4-turbo and our OC prompting achieves high precision in constraint mining with the average of 91.2%. When combining static and dynamic analysis, RBCTest achieves a precision of **78.5%**. RBCTest detects 107 constraints that the dynamic approach misses and 46 more precise (stricter) constraints than the baseline using the AGORA dataset. We also evaluate RBCTest's usefulness by leveraging its generated tests to detect the mismatches between the API specifications and actual execution of the SUTs. A detected mismatch indicates a fault in the SUT or that the specification does not reflect the SUT. We report **21** actual faults found in 8 real-world applications, including 4 issues reported by users on GitLab Forum [1–4]. In brief, this paper makes the following contributions:

1. RBCTest: [A combination of static and dynamic approaches] for constraint mining and test generation for API response bodies by using API specifications.

2. [A manually-verified benchmark] for API testing with respect to response bodies. Our benchmark and code [7] are available for future research on API testing approaches.

3. [An extensive evaluation] showing RBCTest outperforming the state-of-the-art baselines.

2 Motivating Example

2.1 Example and Observations

To illustrate the challenges and motivate our approach, we use Stripe, an online payment service, streamlining the process of charging customers via APIs. Figure 1 shows a simplified description from the API specification (OpenAPI Specification), detailing the `GET` operation for retrieving past charges. This API enables users to retrieve charging records within a specific time interval (lines 11-12). The time interval is defined by the `gt` (greater than) and `lt` (less than) parameters, representing the lower and upper bounds of the time range, respectively. Users can also specify the customer for whom they wish to retrieve charging history (lines 13-16). A successful request returns a response with a status code of `200` and a response body with data. The structure of the response data is outlined in the schema in Figure 1(a), which consists of a list of `charge` objects, each associated with various properties, e.g., amount, created timestamp, currency, and others. For instance, a `GET` request to the `/v1/charges` endpoint with parameters like `'created[gt]=1679090500&customer=cus_idA'` returns a list of charges that satisfy the given conditions. An example response object is displayed in Figure 2. The content of the response is constrained by the actual input parameters in the request. Thus, a complete testing process needs to verify the response content in addition to the returned status code. For example, a test case can check if the `created` field of each returned charge object falls within the specified interval and ensuring that the `customer` field matches the requested ID.

```
(a) charge:
    description: The 'charge' object represents an attempt to move
        money into your account.
    properties:
      amount:
        description: A positive integer can be up to eight digits.
        type: integer
        example: 99999999
      created:
        description: Time at which the object was created. Measured
            in seconds since the Unix epoch.
        type: integer
      currency:
        description: Three lowercase letters.
        type: string
        example: usd
      customer:
        description: ID of the customer this charge is for if existed.
        type: string ...
```

```
(b)paths:
  /v1/charges:
    get:
      description: Returns a list of charges you have created. The
          charges are returned in sorted order ...
      parameters:
        name: created
        description: Only return charges that were created during the
            given date interval.
        schema:
          anyOf:
            - properties:
                gt (integer)
                lt (integer)
        name: customer
        description: Only return charges for the customer specified by
            this customer ID.
        schema:
          type: string ...
```

Figure 1: (a) Schema of a response for 'charge' API in the project Stripe described in a Swagger file and (b) a simplified description for the GET charges API operation from Stripe

```
{
  "id": "ch...15",
  "object": "charge",
  "customer": "cus_id",
  "amount": 1099,
  "created": 1679090539,
  "currency": "usd",...
}
```

Figure 2: A response's body from a `GET` request for Stripe

Observation 1 (Constraints from input parameters). *The response data is constrained by the input parameters from the request. When testing, in addition to verifying the status code, testers need to verify the response data.*

In Figure 1, the descriptions on the properties of the returned `charge` objects define certain constraints on the attributes of those objects. For example, the `amount` property must be a positive integer, with a maximum value of eight digits. The `currency` attribute has a three-letter lowercase code (e.g., `usd`).

Observation 2 (Constraints within response body). *Natural language descriptions on operations express logical constraints on operations, their responses and others, formatting requirements, or value range limitations that must be validated during API testing.*

The Swagger file often includes examples that illustrate specific constraints on data or data types. For instance, in Figure 1(a), the file provides an example of $999,999.99 USD as a positive number for the property `amount`. We can validate the generated test cases against them. An example failing the corresponding test case indicates that the test case—and therefore the mined constraint—may be incorrect.

Observation 3 (Verification with examples). *The illustrating examples in the description can be used to verify against the generated test cases, i.e., the validity of the mined constraints.*

2.2 State-of-the-art Approaches

In API test case generation, validation typically falls into two main categories: status code validation and schema validation.

Status Code Validation: Each HTTP request is returned with a response containing a status code and data. The status code, a 3-digit integer, indicates the outcome of the HTTP request. `2xx` codes signify a successful request. Conversely, `4xx` codes indicate errors, such as a bad syntax request or invalid input values. For instance, in testing the API `'GET/user_information'` with various valid and invalid user IDs, testers would expect the API to return status codes `200` or `404` based on the inputs provided. This validation method is widely used in automation testing tools, e.g., Postman [6] and Katalon [5], RestTestGen [33], or KAT [22].

Schema Validation: Schema validation ensures the response correctness by checking it against a predefined response schema. This verifies the presence of all required properties and the consistency of property data types with their specifications. A lack of required data or a mismatch in data types indicates errors in API services. Tools, e.g., RestTestGen [33], leverage external libraries, such as 'swagger-schema-validator', to facilitate schema validation.

While status code and schema validation effectively cover aspects of data representation and status checking, they may *overlook the logical correctness and validity in the response data*. For instance, if an API request for a charge in 2025 returns one from 2024, or if the charge amount is negative, these issues would not be detected by merely validating the status code or schema.

To address that, the state-of-the-art dynamic approach, AGORA [9] extends Daikon [16], a dynamic instrumenter, to infer the invariants from the values extracted from the execution of the SUT using the APIs. AGORA considers these derived invariants as the constraints for the response bodies. However, inherent from the nature of a dynamic approach, the quality of the derived invariants depends on the values observed during the execution, which might not be diverse enough to reveal the correct constraints. For example, for all the inputs, the charge values might never reach 99,999,999, thus, Daikon returns the maximum charge that is less than that value.

2.3 Key Ideas

From the above observations, we draw the following key ideas to design RBCTest, an approach for mining the constraints of response bodies and then generating test cases to validate them.

Key Idea 1 [Combining Static and Dynamic Approaches to Mine Constraints of API's response bodies]. The first component of RBCTest is a novel LLM-based static approach to mine the constraints for API's response bodies from the OpenAPI Specification (OAS). The constraints on the response data can be found in the specification in either the descriptions of the *operations* (e.g., line 5 of Figure 1(a)) or the descriptions of the *schema for the response data* (lines 2–24 in Figure 1(b)). For example, the description in Figure 1 states that *'the charges are returned in a sorted order with the most recent ones appearing first'*. The schema for the returned values in Figure 1 also provides us several constraints on the *parameters* (lines 5–6) as well as the description of the returned object as a whole (line 2). For example, the amount of a charge is a positive number with up to 8 digits and such value must be of integer.

We integrate AGORA as the dynamic component, leveraging execution data from the SUT to mine invariants for API response bodies. The static and dynamic components complement each other. First, constraint mining remains feasible even if only the API specification or execution data is available. Second, the static component facilitates mining constraints on API response bodies from API specifications, capturing broader constraint information that the dynamic approach might miss. Conversely, runtime data, derived from actual executions, can help refine constraints, providing more precise insights than those typically found in API specifications.

Key Idea 2 [Observation-Confirmation scheme on LLMs for constraints discovery and test generation]. For the task of extracting constraints from API specifications, we utilize the ability of LLMs to comprehend natural language descriptions found in API specifications for constraint mining on the APIs and their parameters. Our experiment showed that direct use of LLMs for constraint mining yields sub-optimal performance. To improve it, we apply an Observation-Confirmation scheme in which the initial result returned from the LLMs will be fed back to themselves in a confirmation prompt to provide better contexts on the constraints.

Key Idea 3 [Generating test cases for the constraints on response bodies]. Our goal is to advance beyond current API testing techniques: we also generate test cases to evaluate the mined constraints. For instance, a test case is generated to verify that the list of charges returned by the API endpoint 'v1/charges' is sorted in reverse chronological order. Another example includes generating a test case to validate the format/value of the returned charge amount.

Key Idea 4 [Filter and Semantic Verifier]. Before asking the LLM to analyze constraints on parameters, we first perform a filtering process to remove invalid constraints. After generating test cases based on the mined constraints, we then introduce a semantic verifier to check these test cases against the examples in the OAS. The rationale is that these examples are typically accurate, as they

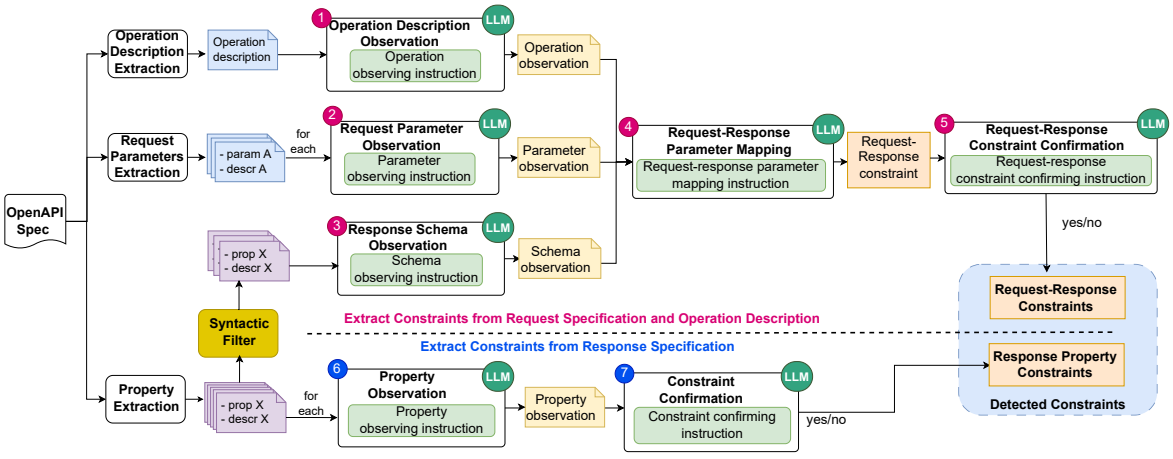

Figure 3: Static Constraint Mining with LLMs

represent the valid data types or values. For instance, an example of $999,999.99 is used to illustrate a positive number for `amount`. We can validate the generated test cases against such examples. If an example does not pass the corresponding test case, it suggests that the test case, and thus the mined constraint, may be incorrect.

3 Static Constraint Mining

To mine the constraints, we observe that the specification for an API endpoint includes two main parts: the *request specification* (the **input** of the API) and *response schema specification* (the **output** of the API). (1) A request specification guides us on how to call an API endpoint, detailing the required *input parameters*, their roles, and the parameters within the request body along with their respective roles. It might also contain the description of the API *operations*. (2) The response schema specification provides instructions on the *response data*, including each property in the response data, its description, datatype, nullability, and other relevant details. These parts contain descriptions that are the targets for constraint mining.

3.1 Constraints from Request Specifications

An essential component of an API specification is the request specification. This part instructs clients on how to correctly initiate API calls, detailing the required inputs and the returned data. Our constraint mining relies on natural language descriptions attached to request parameters or response properties. Since descriptions are optional in OAS, they may appear in one endpoint but not others. To extract them, we first check the current response schema; if none are found, we search the entire OAS document. Unlike KAT [22], which uses LLM-based "description mapping," we avoid potential inaccuracies that could propagate errors. Instead, we adopt descriptions from properties with the same name in other schemas. If none exist, we exclude the property, prioritizing precision over recall.

3.1.1 Request-Response Constraint Mapping. The idea is that to determine a constraint from request parameters, the response data must contain a property that reflects this constraint. Thus, we *identify pairs consisting of a request parameter that includes a constraint and a response data property that reflects this constraint* (Figure 4).

```
PARAMETER_SCHEMA_MAPPING_PROMPT = '''Given a request parameter
      and an API response schema, check if there is a matching
      property in the API response schema.
Request parameter for {method} {endpoint}:
- "{parameter}": "{description}"

Follow these steps to find the matching property: {Instructions
      for chain-of-thought steps}

Schema specification "{schema}": {schema_observation}

Confirm if the request parameter has a matching property in the
      response schema: ...
Identify the corresponding property name of the provided request
      parameter in the schema. ...
If a matching property exists, explain it using this format: ...
```

Figure 4: Request-Response Parameter Mapping Observation

For instance, to verify a "customerID" constraint from Observation 1, the response data should contain a property that identifies the customer. Thus, the required pair is <"customerID", "customer">. If the response data does not have a field to represent a constraint, that constraint is disregarded as it cannot be validated.

```
MAPPING_CONFIRMATION = '''{System prompt}
The request parameter's information:
- Operation: {method} {endpoint}
- Parameter: {parameter_name}
- Description: {description}
The corresponding property's information:
- Resource: {schema}
- Corresponding property: {corresponding_property}
{Instructions for chain-of-thought steps}
Answer format: ...'''
```

Figure 5: Request-Response Constraint Confirmation

3.1.2 Detailed Process. The process of extracting constraints from request parameters encompasses four steps: (1) description extraction, (2) **observation** (1-3 from Figure 3), (3) Request-Response constraint mapping (4), and (4) constraint **confirmation** (5).

Initially, the description extraction phase follows the process shown in Figure 6. An API request comprises multiple parameters and a response data schema. Firstly, a prompt is made to gather observations on the response schema and operations associated

Input:
API_spec(dict): obj of entire API specification.
req_spec(dict): specification obj of a certain request.
resp_schema(dict): schema obj of the associated response.
Output: list of request-response constraint properties

```
1: function getConstrFromReqParas(API_spec, req_spec, resp_schema)
2:   reqRespConstraint ← []
3:   respSchemaObser ← LLM().respSchemaObser(resp_schema)
4:   operationObservation ← LLM().operationObser(desc)
5:   for each param in req_spec do
6:     desc ← req_spec[param]["desc"]
7:     if req_spec[param]["desc"] = NULL then
8:       desc ← findExactMatchParameter(API_spec, param)
9:       if desc = NULL then
10:        # Skip this param
11:        continue
12:    # Use LLM to find constraint for this param
13:    paramObser ← LLM().parameterObser(param, desc)
14:    answer, corrProp, explain ← LLM().reqRespMapping
         (param, desc, paramObser, respSchemaObser)
15:    if answer = TRUE then
16:      confirmation ← LLM().
   confirmReqRespMapping(param, corrProp, explain)
17:      if confirmation = TRUE then
18:        reqRespConstraint.append((param, corrProp))
19:  return reqRespConstraint
20: end function =0
```

Figure 6: Extract Constraints from Request Parameters

with this request (lines 3–4, Figure 6). For each request parameter, we engage LLM to provide its observations on that parameter by presenting it alongside its description (line 13, Figure 6). The LLM is expected to provide a description of any constraint within this parameter description. If such description is missing, the specification is searched for another parameter with an identical name.

These descriptions support the next step: Request-Response parameter mapping (Block 4, line 14). We guide LLM through a two-step reasoning process using a tailored prompt (Figure 4). First, LLM receives a brief description of the request parameter, including its details and observations (2), ensuring a clear understanding of its intent and constraints. Second, we present the response schema, observations from Block 3, and ask LLM to identify a matching property. A match occurs when the request parameter filters response data or both share the same value meaning.

From this two-step reasoning, we require LLM to answer three questions: (1) Is there a matching property in the response schema? (2) What is this property? (3) How does the request parameter influence this property?

If the answer to (1) is negative, indicating no property reflects this request parameter, we disregard this parameter. Otherwise, we proceed to the final prompt (Figure 5), which is the confirmation of the mapping (lines 15–18, Block 5). In this prompt, we present the pair of the request parameter and the matched property from Block 4 and ask LLM to confirm the accuracy of this mapping (line 16). This step aims to minimize the occurrence of false positives. Finally, the validated pairs of request parameters and response properties are stored as Request-Response constraints (lines 17–18).

Input:
API_spec(dict): obj of entire API specification.
response_schema(dict): schema obj of a certain response.
knowledge_base(dict): gained knowledge.
Output: list of constraint properties.

```
1: function getConstraintInsideResponseSchema(API_spec, re-
   sponse_schema, knowledge_base)
2:   constraint_properties ← []
3:   for each prop in response_schema do
4:     description ← response_schema[prop]["description"]
5:     if description = NULL then
6:       description ← exactMatchProp(API_spec, prop)
7:       if description = NULL then
8:         # Skip this property
9:         continue
10:    if prop in knowledge_base then
11:      if knowledge_base[prop] = TRUE then
12:        constraint_properties.append(prop)
13:    else
14:      datatype ← response_schema[prop]["datatype"]
15:      prop_obser ← LLM().propertyObser(prop, datatype, descrip-
   tion)
16:      constraint_confirmation ← LLM(). constraint_ confirmation
   (prop, datatype, description, prop_obser)
17:      if constraint_confirmation = TRUE then
18:        constraint_properties.append(prop)
19:      # Add this property to knowledge base
20:      knowledge_base[prop] ← constraint_confirmation
21:  return constraint_properties
22: end function =0
```

Figure 7: Extract Constraints from Response Specifications

3.2 Constraints from Response Specifications

First, we examine descriptions within the response schema specification, as they provide direct constraints by mapping each property. Each endpoint includes a response schema that guides clients on parsing returned data, structuring the response object with properties and descriptions (Figure 2). Our constraint-mining algorithm, leveraging LLM, is outlined in Figure 7. It first extracts descriptions for each property (lines 4-8) following Section 3.1. If a description exists, we check a knowledge base of LLM-identified constraints to avoid redundant queries. If found, we reuse the stored information; otherwise, we prompt LLM to extract constraints.

Observation-Confirmation Strategy. This process involves two phases: *observation* (line 15) and *confirmation* (line 16). Our experiments reveal that when descriptions lack detail for constraint extraction, LLM may resort to fabricating details, a phenomenon known as hallucination. Drawing inspiration from the Chain-of-thought [34], we divide the task of extracting constraints into two phases of **observation** and **confirmation**, the initial prompt better contextualizes the description of constraints, enabling the subsequent prompt to more accurately determine a constraint.

In the observation phase (Block 6), LLM is prompted to identify constraints from the description. For example, given a property `date` (type: `string`) described as *"ISO date: the literal date of the holiday"*, LLM might infer: *"The date must follow the YYYY-MM-DD format for validity, ensuring consistency within the API."* This adds specificity not explicitly mentioned in the description.

Figure 8: Example outputs from RBCTest and AGORA.

ID	Constraints	Invariants
1	the number of returned items has to be less than or equal to the requested limit	input.limit >= size(return.items[])
2	–	return.total >= size(return.items[])
3	return.total is an integer larger than or equal to 1	return.total >= 1
4		return.total is Integer
5	–	input.market is a substring of return.href
6	–	input.id is a substring of return.href
7	number of adult guests (1-9) per room	return.adults is Integer
8	return.price must be within input.price_range	–
9	An amount is a positive integer can be up to eight digits	–
		–

Next, the observation is fed into the Constraint Confirmation Prompt (Block 7), where LLM validates whether the extracted constraint provides enough detail for script generation. This step, similar to Figure 5, ensures constraints specify values, ranges, or formats. If confirmed, the constraint is marked as a Response Property constraint and stored in the knowledge base.

4 Combining Constraints and Invariants

Constraints detected by our static method are based on OAS while invariants determined by AGORA come from execution data. Let us present our ensemble approach to combine the constraints and invariants. Figure 8 shows sample outputs produced by each method. Several constraints can only be identified using the OAS. For instance, Constraint 9 in Figure 8, which specifies that the charge amount must be smaller than 99,999,999, can only be extracted from the OAS. This is because AGORA requires a sufficiently large number of API executions to infer such constraints. Conversely, some constraints can only be inferred at run-time, e.g., those related to returned Hrefs—URLs containing requested information.

In AGORA, each detected invariant is associated with a set of variables, as depicted in Figure 8, and a given set of variables may be linked to one or more invariants (e.g. Invariants 3-4). In our static method, each constraint is tied to a specific set of variables, which aligns with how AGORA groups invariants. Because each constraint is modeled via textual representations, we leverage an LLM and its natural-language text understanding to derive the relevant variables in the constraint.

If a constraint and an invariant involve different sets of variables, we include both in the resulting constraints for RBCTest, as they are uniquely detected by each approach. If they involve the same, we select the constraint or invariant with the stricter condition. For instance, in Figure 8, row 7, we chose the constraint identified by our static method, as it is stricter than the invariant detected by AGORA. To determine which is stricter, we leverage an LLM, which can interpret well the conditions expressed in natural language.

5 Constraint Test Generation

Constraint tests are generated using LLM (Figure 10) based on two types of constraints: Request-Response constraints and Response Property constraints. Request-Response constraints stem from request parameter descriptions, while Response Property constraints are derived from the response schema. These tests take a response body and request details as input, producing outcomes of 'matched,' 'mismatched,' or 'unknown.' A 'matched' outcome confirms that the response satisfies the constraint, whereas 'mismatched' indicates a violation. An 'unknown' outcome suggests the absence of the relevant property, possibly due to its optional nature in the specification. We also incorporate a semantic verifier that cross-checks test cases against examples in the OAS file. If a valid example fails a generated test, the test is deemed incorrect.

```
CONSTRAINT_TEST_GEN_PROMPT = '''
Generate a Python script to check if a property in a REST API's
      response meets specified constraints and rules.
Constraint description:
- Constraint from request parameter: {parameter}
- Constraint description: {constraint_description}
API response schema: {response_schema_specification}
The property of the provided request parameter in API response:
- "{property}": "{prop_description}"
Based on the provided constraint from request param, and the
      respective attribute in the API response, generate a Python
      script to verify the '{property}' property in the response.
Rules: {Rules for test gen}
Format the script as shown: ...
```

Figure 9: Constraint Test Generation Prompts

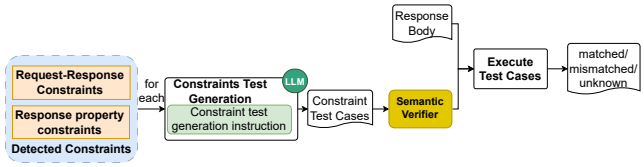

Figure 10: Constraint Test Generation in RBCTest

5.1 Request-Response Constraints Testing

The Request-Response Constraint Test identifies dependencies between a constrained request parameter and its corresponding response property. To generate a test case, we use the prompt in Figure 9, which requires four inputs: the parameter name, its constraint description, the corresponding property name, and the response schema. LLM generates a validation function with two inputs: *the response body* and *the request parameter*. It then creates a script to check conditions between them. For instance, when validating a 'created' time interval (Observation 1), LLM extracts the 'created' time from the response body and the conditional values from the request parameter ('created[gte]', 'created[lte]') before performing logical comparisons. To ensure robustness, we guide LLM with predefined rules: using a `try-catch` block for error handling, excluding examples, and following a standardized function template. These rules ensure consistency and focus on constraint verification.

5.2 Response Property Constraints Testing

The Response Property constraint Test directly correlates the property's description with the property itself. To create a test case for this constraint, we utilize a specific prompt (Figure 9). This prompt requires three inputs: the property name, the constraint description, and the response data schema. The description extracted from the

previous mining step gives the necessary information for generating constraint verification code, while the response data schema defines the structure and type of the expected data. This schema guides LLM in generating code to parse response data. We guide LLM to follow predefined rules (similar to 5.1) to maintain consistency in the generated code in different constraints.

6 Empirical Evaluation

For evaluation, we seek to answer the following questions:

RQ1. [Constraints Test Generation Accuracy] How well does RBCTest perform in comparison with individual static and dynamic approach in generating test cases from the mined constraints?

RQ2. [Constraints Mining Accuracy] How well does RBCTest perform in detecting constraints in the RESTful API specification?

RQ3. [Accuracy in Test Generation from the Correctly Mined Constraints] How accurate is our approach in generating test cases for mined constraints?

RQ4. [Usefulness in Response Body Testing] How accurate is our approach in detecting mismatches between the specification and its working APIs?

RQ5. [Ablation Study] How much Confirmation-Observation prompting and Semantic Verifier contribute to its performance?

Data Collection. We curated a dataset from eight real-world services, including GitLab and Stripe, comprising 59 endpoints and 83 operations. These services were selected for several reasons: (1) They feature complex request-response structures across diverse business domains. (2) They have been widely used in API testing research, such as 'Canada Holidays' [22], GitLab-services [14, 18, 22, 23, 35, 36], and 'Stripe' [22, 28, 30]. (3) Their active status allows API calls to collect real response data. (4) Their specifications vary in documentation quality, enabling evaluation across different levels of completeness. We selected *API endpoints* based on the presence of parameter or response descriptions, excluding those without any descriptions, as constraints could not be identified. Stripe, offering a test mode with limited endpoints, contains deeply nested response schemas, often missing values for validation. To mitigate this, we included only Stripe endpoints without nested schemas, retaining 7 in total. We cover 3 REST methods: GET, PUT, and POST (Table 3).

7 Experimental Results

7.1 Constraints Test Generation Accuracy (RQ1)

7.1.1 Methodology. We evaluated the entire process in our LLM-based static method. Specifically, we manually evaluated the generated test cases produced by the entire process from mining constraints and test case generation from mined constraints.

We used two datasets: 1) the AGORA [9] dataset, and 2) a self-collected dataset (will be explained later). For comparison, we grouped the invariants from AGORA into the groups for one specific variable, two variables, and so on. We similarly grouped the mined constraints from our LLM-based static method and compared the groups on the same set of variables from both approaches.

As evaluation metrics, we report the number of detected constraints, the number of True Positives (TP), False Positives (FP), and the precision P, with $\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP+FP}}$. Specifically, for AGORA, we reuse their experimental results (Total, TP, and P) on their dataset.

Table 1: Constraints Test Generation on AGORA dataset.

	Static			Dynamic			Unique		Overlapping		
APIs	**Total**	**TP**	**P**	**Total**	**TP**	**P**	**Static**	**Dynamic**	**+S**	**+D**	**Eq.**
A.Hotel	44	39	88.6	117	61	52.1	20	34	8	1	10
GitHub'	16	13	81.3	198	194	98	3	175	6	0	4
GitHub"	7	7	100	150	127	84.7	7	121	0	0	0
Marvel	28	23	82.1	115	55	47.8	13	35	6	0	4
OMDB	4	4	100	16	15	93.8	3	13	0	0	1
OMDB'	2	1	50	5	5	100	0	3	0	0	1
Spotify	21	21	100	41	41	100	11	26	3	2	5
Spotify'	13	12	92.3	68	58	85.3	1	29	5	4	2
Spotify"	15	15	100	55	45	81.8	3	28	6	2	4
Yelp	2	1	50	30	12	40	1	11	0	0	0
YouTube	65	62	95.4	194	111	57.2	45	52	12	2	3
TOTAL	**217**	**198**	**91.2**	**989**	**724**	**73.2**	**107**	**527**	**46**	**11**	**34**

Static: LLM-based, Dynamic: Execution-based (AGORA), +S: Static constraint better, +D: Dynamic constraint better, Eq: Equivalent.

7.1.2 Results on AGORA dataset. Table 1 shows the results on the AGORA dataset with 11 API operations on 7 APIs. Our LLM-based static method identified 217 constraints, with 198 true positives, resulting in a precision of 91.2%. In contrast, AGORA detected 724 true positives out of 989 invariants, resulting a precision of 73.2%.

RBCTest combines the constraint results from both LLM-based static method and the dynamic approach in AGORA. Thus, we also conducted an overlapping analysis between them. Overlapping constraints are those applied to the same variables, and we compare how well the constraints are detected. A constraint is considered as better than one invariant or a group of invariants if it refers to a narrower set of values for the variable(s) while still adhering to the variables' description. Similar definition is used for a better invariant or group of invariants. They are equivalent if they cover exactly the same set of possible values for the variable(s).

Our overlapping analysis results reveal that *107 constraints were uniquely detected by our LLM-based static method, while 527 were exclusively identified by AGORA*. We further investigated the invariants detected only by AGORA and found that 319 of them pertained to variables lacking descriptions in API specification. This suggests that these invariants were only detectable through the API responses. These invariants primarily involved checks such as 1) whether a variable is `URL` (33%), 2) a substring of another variable (32%), 3) equal to another variable (13%), or 4) related to string length (7%), and (5) 15.6% covering other types.

Conversely, *the constraints uniquely detected by our LLM-based static method* mainly occurred in scenarios where the API specification provided variable descriptions, but the AGORA dataset's API responses did not include these variables (often optional fields). AGORA's dependency on the diversity of API responses at runtime limits its detection capability in such cases. For instance, in the YouTube API, there are 14 distinct rating schemas that appear only in specific request regions. If AGORA's API calls do not cover all regions, these schemas remain undetected.

7.1.3 Result Analysis. A closer look at the cases where our LLM-based static method was superior reveals two main reasons for its better performance. First, our LLM-based static method *handled more specific domains or ranges of values*. For example, in the Amadeus Hotel API, the `roomQuantity` value is specified as *"an integer between 1 and 9."* Our LLM-based static method correctly identified this constraint and generated an appropriate test, whereas AGORA

Table 2: RBCTest: Combining constraints and invariants

API-Op	Static	Dyn.	Unq. S	Unq. D	Dist.	Overlapping		Combination		
						Cons.	Incons.	Total	TP	P%
A.Hotel	39	61	20	34	54	11	8	109	73	67
GitHub	13	194	3	175	178	4	6	194	188	96.9
GitHub'	7	127	7	121	128	0	0	152	128	84.2
Marvel	23	55	13	35	48	4	6	113	58	51.3
OMDB	4	15	3	13	16	1	0	18	17	94.4
OMDB'	1	5	0	3	3	1	0	6	4	66.7
Spotify	21	41	6	35	41	7	3	47	47	100
Spotify'	12	58	1	29	30	6	5	50	41	82
Spotify"	15	45	1	11	12	6	6	53	43	81.1
Yelp	1	4	1	11	12	0	0	31	12	38.7
YouTube	62	111	45	52	97	5	12	151	114	75.5
Total	**198**	**724**	**107**	**527**	**634**	**45**	**46**	**924**	**725**	**78.5**

Unq: Uniquely detected, Dist: Distinct, (In)Cons: (In)Consistent.

provided a general invariant "Numeric," encompassing any integer, float, etc. Similarly, in the Spotify API, AGORA expected the `thumbnailHeight` to be *"one of (64, 300, 640),"* based on the observed data at runtime, but our LLM-based static method correctly identified it as *"image height in pixels,"* which implies any positive integer. Second, our LLM-based static method excelled in mining specific constraints. For instance, in the Amadeus Hotel API, the `sellingTotal` is defined as *"= Total + margins + markup + totalFees - discounts."* While AGORA simply concluded that `sellingTotal` was numeric, our LLM-based static method was able to be more specific in the constraint mined from the specification. However, there were a few cases where AGORA outperformed our LLM-based static method. Those cases often involve invariants verifying the format of variables containing URL, and our LLM-based static method sometimes treated URLs as mere strings without further validation.

7.1.4 Combined method. From Table 2, we can see that combining both static and dynamic approaches yield a better result than that of individual method. In total, RBCTest has 107 constraints detected only by our static method, 527 detected only by AGORA. For cases where *both approaches detected constraints and invariants* on the same variables, in total, we identified 91 overlapping constraints and invariants. Among them, 46 cases are inconsistent. Inconsistencies occur where an invariant extracted from execution data defines a set of instances outside of the set defined by the respective constraints mined from the specification. The inconsistencies detected by RBCTest reflect a coding bug or out-of-date specification.

As seen in Table 2, RBCTest (the combined method) detects a total of 924 constraints, of which 725 are true positives, i.e., 78.5% precision. RBCTest can identify constraints from both views while maintaining sufficient precision. *Despite of lower precision than the LLM-based static method, RBCTest detects more true positive constraints (725) in comparison to the individual methods, that is, the 198 true positive constraints from the LLM-based static method and the 618 grouped invariants from the dynamic approach, AGORA.*

7.1.5 Results on RBCTest dataset. For more generalization, we repeated the above experiment on our collected dataset whose statistics are shown in Table 3. Due to certain services requiring enterprise subscriptions, we were unable to execute AGORA on this dataset. In general, our dataset has 8 APIs with 83 operations,

Table 3: Constraints Test Generation on RBCTest Dataset

API	# Op.	Methods	Type	GT	RBCTest					
					TP	FP	FN	P%	R%	F1%
C.Holiday	4	GET	RP	24	16	0	8	100	66.7	80
G.Branch	5	GET,POST	RR	49	35	3	14	92.1	71.4	80.4
G.Commit	11	GET,POST,PUT	RR	73	54	3	19	94.7	74	83.1
G.Groups	14	GET,POST,PUT	RR	85	61	3	24	95.3	71.8	81.9
G.Issues	21	GET,POST,PUT	RR	141	92	3	49	96.8	65.2	77.9
G.Project	15	GET,POST,PUT	RR	144	110	11	34	90.9	76.4	83
G.Repo.	3	GET,POST	RR	44	33	3	11	91.7	75	82.5
Stripe	10	GET,POST	RR	19	12	1	7	92.3	63.2	75
			RP	21	16	1	5	94.1	76.2	84.2
Total	83			600	429	28	171	93.9	71.5	81.2
Stdev								2.9	4.9	2.9

Services: Canada Holidays, GitLab {Branch, Commit, Groups, Issues, Project, Repository}, and Stripe. # of Operations (No. Ops), # of ground truth constraints (GT), Precision (P), and Recall (R). RR for Request-Response Constraints, and RP for Response-Property Constraints.

Table 4: LLM-based Constraint Mining, Constraints Test Generation, and Test Outcomes on RBCTest Dataset.

API	Type	Constraints Mining (RQ2)						Test Gen. (RQ3)			Test Out. (RQ4)		
		TP	FP	FN	P%	R%	F1%	N	✓	P%	✓	×	?
C.Holiday	**RP**	16	0	8	100	66.7	80	16	16	100	12	0	4
G.Branch	***RR***	36	2	13	94.7	73.5	82.8	36	35	97.2	33	2	0
G.Commit	***RR***	55	2	18	96.5	75.3	84.6	55	54	98.2	40	8	6
G.Groups	***RR***	63	2	22	96.9	74.1	84	62	61	98.4	60	1	0
G.Issues	***RR***	93	2	48	97.9	66	78.8	93	92	98.9	80	7	5
G.Project	***RR***	110	11	34	90.9	76.4	83	110	110	100	102	0	8
G.Repo.	***RR***	33	3	11	91.7	75	82.5	33	33	100	30	2	1
Stripe	***RR***	12	1	7	92.3	63.2	75	12	12	100	7	0	5
	RP	17	0	4	100	81	89.5	17	16	94.1	14	1	1
Total		**435**	**23**	**165**	**95.7**	**72.4**	**82.2**	**434**	**429**	**98.8**	**378**	**21**	**30**

For test outcomes: ✓ (Matched), × (Mismatched), and **?** (Unknown).

covering the 'GET', 'POST', and 'PUT' methods. We manually reviewed the API specifications of these services and identified a set of 600 correct constraints as the ground truth.

As seen in Table 3, within this dataset, RBCTest via LLM-based static method (no AGORA) identified 457 constraints, corresponding to 457 test cases generated by our LLM-based static method. This includes 28 false positives and 171 missed constraints, yielding an overall **precision of 93.9%** and **a recall of 71.5%**, with an **F1 score of 81.2%**. All standard deviations were below 5%, indicating consistently strong performance across different APIs. The inaccuracy mainly arises from the effects of the filter and verifier: in an effort to minimize the potential for erroneous outputs from the LLM, we restricted both the input to and the output from the model to ensure accuracy. This result is consistent with that in Section 7.1.2.

Our analysis reveals that *response-property constraints*—which apply to a single response property—are typically straightforward, focusing on format or value range. They account for 32 of the 429 detected constraints. However, in GitLab services, they were difficult to detect due to sparse property descriptions. In contrast, *request-response constraints* were more prevalent, as they involve multiple variables, capturing complex relationships where a request parameter can influence multiple response properties.

7.2 Constraint Mining Accuracy (RQ2)

7.2.1 Methodology. In RQ1, we assess the entire process from the input of API specifications to the test generation. In this RQ2, we

focus on RBCTest's first component, Static Constraint Mining via LLMs. We used the RBCTest dataset with the available API specifications. Thus, we did not run AGORA. To decide the constraints' correctness in natural language, we use the following rules:

(1) Request-Response Property constraint: This type of constraint on a request parameter must correspond to a property in the response data that reflects this constraint.

(2) Range-of-Value constraint: The type of constraint must specify all possible values or provide a specific data range.

(3) Data Format constraint: This must describe the constraints on data format or refer to widely used formats (e.g., ISO, Unix).

7.2.2 Results. As seen in Table 4, overall, 95.7% of the constraints identified by our LLM-based static method are valid, although it missed 165 out of 600 constraints noted in the ground truth. All metrics are notably higher than those of the entire process evaluated in RQ1, as in RQ1's experiment, we aim to account for the entire process including both the validity of constraints and the accuracy of the generated tests. As outlined in Section 3, the detected constraints fall into two categories: Response Property constraints and Request-Response constraints. Our LLM-based static method's constraint mining for Response Properties proves more effective than for Request-Response constraints, achieving 100% precision and 73.3% recall compared to 94.6% precision and 72.4% recall for Request-Response constraints.

Response Property constraints primarily define format and value ranges, with clear descriptions like "three-letter currency code" or "Unix epoch timestamp." In contrast, *Request-Response constraints* are more complex, requiring precise mappings between request parameters and response properties. For example, in GitLab, 'since' and 'until' correctly map to 'created_at' to enforce range conditions. However, most false positives arise from incorrect mappings, especially in GitLab, where response body descriptions are lacking. Without details, mappings rely on attribute names, leading to errors—e.g., the 'avatar' parameter (for uploading images) is mistakenly linked to 'avatar_url' in the response due to name similarity.

7.3 Test Generation Accuracy from Correctly Mined Constraints (RQ3)

7.3.1 Methodology. While RQ1 evaluates the entire process from constraint mining to test generation, this RQ3 focuses only on evaluating the test generation component for the constraints *correctly detected by RBCTest via our LLM-based mining on the RBCTest dataset (thus, we did not run AGORA).* We performed test generation from those correct constraints. The generated test cases are manually evaluated if they correctly verify the associated constraints.

We used the following rules to check if a test case is correct:

(1) Test Input: The generated test case is correct if it correctly receives two inputs for Request-Response constraints: 1) requested information and 2) response data, and one input for Response Property constraints: response data.

(2) Constraint Handling: The generated test case must cover all conditions in the constraint.

(3) Test Output: The test must return:

i) 0 (`unknown`) if lacking of sufficient data for condition checking (e.g., empty or `null` value).

ii) 1 (`matched`) if the provided input satisfies the constraint.

iii) -1 (`mismatched`) if the provided input does not satisfy it.

We only consider the set of test cases derived from valid constraints as identified in RQ2 (denoted as N in Table 4).

7.3.2 Results. As shown in Table 4, RBCTest generated 434 test cases for 435 TP constraints (one test case was discarded by the verifier), including 401 for Request-Response constraints and 33 for Response Property constraints. It achieves a precision of **98.8%** across 8 services and confirmed 429 correct test cases. Examining services, 4/8 services achieved a precision of 100%, while the lowest precision was in the Stripe service, with a precision of 94.1%.

Results indicate that RBCTest excels in generating code for *Response Property constraints*, which mainly involve format or value range validation. In contrast, *Request-Response constraints* are more error-prone due to the complex logic required to parse and verify dependencies between request parameters and response properties. Further analysis reveals that test generation errors primarily stem from (1) missing descriptions and (2) ambiguous keywords.

Consider the `'get-/issues'` endpoint in GitLab Issues, where a constraint exists between the request parameter `'due_date'` and a response property of the same name. The parameter is described as *"Accepts: 0 (no due date), overdue, week, month, next_month,"* while the response property lacks a description. As a result, our tool generates test cases to validate the property against this constraint, even though `'due_date'` in the response is a date-time string, leading to incorrect tests. Such errors are common in GitLab due to insufficient descriptions for response properties.

In the same `'get-/issues'` endpoint, the `'milestone'` parameter is described as *"The milestone title. None lists all issues with no milestone. Any lists all issues that have an assigned milestone."* The API filters returned issues based on `'None'` or `'Any'` from the request parameter. However, RBCTest misinterpreted `'None'` as Python's `null`, leading to errors. This mistake in `'milestone'` propagated to 13 other incorrect test cases due to its involvement in multiple operations.

7.4 Usefulness: Detecting Mismatches between Constraints and Response Bodies (RQ4)

7.4.1 Methodology. We use the generated test cases to *detect mismatches between the constraints in the API specification and the API response bodies.* With this goal, we did not run AGORA. For each API endpoint, we execute the API with multiple sets of request parameters. For each API execution, we collect 1) request information (i.e., *what is expected* from the API call), and 2) response data (i.e., the *actual response*). Response data contains multiple properties, each attached with constraints and generated constraint test cases. We ran the test cases associated with the response data to collect the outcomes. We also ran the inputs used in AGORA to verify against our test cases of correctly generated constraints. If the result is false, we report a mismatch. Otherwise, it is a match.

7.4.2 Results. We performed test runs on 429 correctly generated tests from RQ3 (Table 4). Our test results indicate 378 'matched' response bodies (i.e., consistent with the specification), 21 'mismatched', and 30 'unknown'. Out of 434 correctly mined constraints, 378 were verified by the tests, meaning that 87.1% of the constraints are met by the actual execution of the SUTs. Our tool detected

Table 5: Contribution of components in RBCTest (RQ5)

Variant	Constraints Mining					
	TP	FP	FN	P	R	F1
RBCTest	435	22	165	95.2	72.5	82.3
RBCTest$^-$	197	127	403	60.8	32.8	42.6

21 mismatches, revealing inconsistencies between specifications and execution of the APIs. An 'unknown' occurs when a property is absent in the response body due to its optional nature.

7.4.3 Analysis. We revealed the following root causes of these mismatches: (1) incompatible data formats, (2) not-explicitly-described nullable properties, and (3) inter-parameter request dependencies.

(1) Incompatible Data Formats: Our tool **detected 21 constraint mismatches**, mainly from GitLab services. For instance, the constraint *"date will be returned in ISO 8601 format* `YYYY-MM-DDTHH:MM"` appears in GitLab operations. However, the actual data format is `"2012-09-20T08:50:22.000Z"`, which includes a decimal part for seconds, leading to an inconsistency. Interestingly, we found that three instances of this type of inconsistency **were reported as the issues** on the GitLab forum [1, 3, 4]. This is *anecdotal evidence on RBCTest's usefulness in detecting real-world issues.*

(2) Not-Explicitly-Described, Nullable Properties: This issue is common in GitLab, where some response properties are nullable but lack descriptions in the specification, leading to parsing errors. Notably, these issues have been reported on the GitLab forum [2].

(3) Inter-Parameter Request Dependencies: This was found in only one case. For the operation `'GET/groups'` from GitLab Group, there is a constraint on the request parameter `"order_by"` affecting the `"name"` property in the response data. This logic dictates that the array of groups in the API response should be sorted according to the `"order_by"` parameter. RBCTest checked only one-to-one conditions among parameters, whereas the sorting order depends on both `"order_by"` and `"sort"` parameters (specifying the sort direction). As a result, this resulted in a detected mismatch.

7.5 Ablation Study (RQ5)

In this experiment, we first built a variant that does not contain both observation-confirmation and semantic verifier components. To learn more the static method, the dynamic component was excluded. We modify the prompt in Figure 4 as follows for constraint mining. Instead of providing GPT-4-Turbo with observations derived from another prompt, we feed the description of the parameters and response schema as in the API specification. This prompt replaces Blocks 1-5 from Figure 3. After providing data on a property, we instruct the LLM to decide if the given property contains a constraint and if there is enough to verify it. This modification merges steps 6 and 7 into a single step. The output is *yes* or *no*.

As seen in Table 5, the elimination of observation-confirmation prompting and semantic verifier affects the outcomes, as evidenced by a reduction of 238 correct constraints. Concurrently, there is an increase in the number of false positives. The F1-score of RBCTest$^-$ is reduced by half. False positives frequently arise when the model mis-mapping the request parameters with the response properties based on their names. Such mistakes are less common in RBCTest, where the observation prompt is used to enhance the property information before it is processed by the confirmation prompt. *This result confirms our key contribution of our LLM prompting strategy.*

Regarding the semantic verifier, its goal is to remove the invalid constraints to improve precision. we currently used a simple verifying mechanism via examples in API specification (Section 2.3). We removed the semantic verifier and ran the static component on two datasets. In RBCTesting dataset with 89 examples, one example invalidated one detected constraint. In AGORA dataset, with 223 examples over 11 API operations, 6 examples were able to invalidate 6 false-positive constraints. We reported that the verifier accurately preserves all valid constraints and successfully eliminates 7 incorrect constraints, thus achieving higher precision (89.0% increasing to 91.2%) while maintaining recall (72.5%). This results show that more examples in the API specification might help invalidate more incorrect constraints and confirm the correct ones. Moreover, other types of semantic verifier can be integrated into our framework such as constraint solvers, SMT solvers, domain-specific checkers (valid zip code, phone number format, or valid date checkers), etc.

8 Threats to Validity

1. Internal Validity. LLM hallucinations might lead to unexpected outcomes. To mitigate this, we incorporated (1) a semantic verifier (Figure 10) and (2) observation-confirmation prompting (Figure 3), which enhance validation through external API specifications and internal consistency checks. Our ablation study confirmed that these methods have a positive impact on performance. We segmented specifications into small sections (Figure 3) to reduce context window, minimizing hallucinations and improving accuracy.

2. External Validity. Our dataset may not fully represent the API landscape, affecting generalizability. Outcomes could vary with different datasets, particularly those with unique constraints. Generated test cases may miss general constraints or edge cases beyond what is explicitly documented. Implicit constraints not in API specifications might also be valid but unaccounted for. External tools may introduce inaccuracies, potentially affecting the results.

9 Related Work

Recent surveys on API testing [15, 17, 21, 25, 26, 28, 32] reveal a trend towards automation adoption. AI/ML are used to enhance various aspects of API testing including of generating test cases [14, 27, 33], realistic test inputs [8], and identify defects early in the development [11–13, 24, 31, 37–40]. AGORA is a dynamic approach [9].

The advances of LLMs have impacted API testing. [22] leverages the language capabilities of GPT to fully automate API testing using only an input OpenAPI specification. It builds a dependency graph among API operations, generating test scripts and inputs. Moreover, [20] applied GPT to augment specifications, enriching them with explanations of rules and example inputs. Before the era of LLMs, ARTE [8] aimed to generate test inputs for API testing, employing NLP. Morest [24] introduced a model-based RESTful API testing method using a dynamically updating RESTful-service Property Graph, showing improvements in code coverage and bug detection.

RESTler [14] is a stateful fuzzing tool of REST APIs, which analyzed specifications, inferred dependencies among request types, and dynamically generated tests guided by feedback from service

responses using a test-generation grammar. Similarly, RestTestGen [33] applied specifications for automatically generating test cases, checking both response status and response data schemas. NLPtoREST [19] used the NLP technique Word2Vec [29] to extract rules from human-readable descriptions in an OpenAPI specification, enhance, and add them back to the specification.

10 Conclusion

This paper presents RBCTest, a combination of an LLM-based static method and a dynamic method for mining constraints from API response bodies. We found that the two approaches complement well, leading to RBCTest's high precision of 78.5%. Specifically, LLMs with Observation-Confirmation prompting achieve high precision in constraint mining of 95.7%. Despite a lower combined precision, RBCTest detects more true positive constraints. It also generates test cases to validate these constraints. Our test cases detected 21 mismatches in real-world APIs. Our reported mismatches were actually confirmed by developers in their forums.

When combining static and dynamic approaches, RBCTest takes advantage of both LLM's capabilities to comprehend natural language in API specifications, when available, and API execution information to detect constraints on response bodies. This method allows software teams to test API services of SUTs in both the development and evolution stages. By using API specifications, RBCTest supports applicable use cases where developers need to investigate and test third-party APIs with publicly available specifications before using them for their applications.

References

[1] 2024. GitLab Issue 19667. https://gitlab.com/gitlab-org/gitlab/-/issues/19667
[2] 2024. GitLab Issue 348376. https://gitlab.com/gitlab-org/gitlab/-/issues/348376
[3] 2024. GitLab Issue 349664. https://gitlab.com/gitlab-org/gitlab/-/issues/349664
[4] 2024. GitLab Issue 410638. https://gitlab.com/gitlab-org/gitlab/-/issues/410638
[5] 2024. Katalon Studio Automation Testing Tool. https://katalon.com/
[6] 2024. Postman API Testing Tool. https://www.postman.com/
[7] 2024. RBCTest. https://github.com/api-rbtest/api-rbtest-public-code
[8] Juan C Alonso, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* 49, 1 (2022), 348–363.
[9] Juan C. Alonso, Sergio Segura, and Antonio Ruiz-Cortés. 2023. AGORA: Automated Generation of Test Oracles for REST APIs. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1018–1030. https://doi.org/10.1145/3597926.3598114
[10] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 1–37.
[11] Andrea Arcuri. 2020. Automated black-and white-box testing of restful apis with evomaster. *IEEE Software* 38, 3 (2020), 72–78.
[12] Andrea Arcuri and Juan P Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–31.
[13] Andrea Arcuri and Juan P Galeotti. 2021. Enhancing search-based testing with testability transformations for existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–34.
[14] 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.
[15] Adeel Ehsan, Mohammed Ahmad ME Abuhaliqa, Cagatay Catal, and Deepti Mishra. 2022. RESTful API testing methodologies: Rationale, challenges, and solution directions. *Applied Sciences* 12, 9 (2022), 4369.
[16] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1–3 (dec 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015
[17] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2022. Testing RESTful APIs: A Survey. *ACM Transactions on Software Engineering and Methodology* (2022).
[18] Stefan Karlsson, Adnan Čaušević, and Daniel 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)*. IEEE, 131–141.
[19] 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.
[20] 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.
[21] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. 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*. 289–301.
[22] Tri Le, Thien Tran, Duy Cao, Vy Le, Vu Nguyen, and Tien N. Nguyen. 2024. KAT: Dependency-aware Automated API Testing with Large Language Models. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE.
[23] Jiaxian Lin, Tianyu Li, Yang Chen, Guangsheng Wei, Jiadong Lin, Sen Zhang, and Hui Xu. 2022. foREST: A Tree-based Approach for Fuzzing RESTful APIs. *arXiv preprint arXiv:2203.02906* (2022).
[24] 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.
[25] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the faults found in rest apis by automated test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–43.
[26] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-box and white-box test case generation for RESTful APIs: Enemies or allies?. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 231–241.
[27] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*. Association for Computing Machinery.

[28] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2022. Online testing of RESTful APIs: Promises and challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 408–420.

[29] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL] https://arxiv.org/abs/1301.3781

[30] A Giuliano Mirabella, Alberto Martin-Lopez, Sergio Segura, Luis Valencia-Cabrera, and Antonio Ruiz-Cortés. 2021. Deep learning-based prediction of test input validity for restful apis. In *2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*. IEEE, 9–16.

[31] Omur Sahin and Bahriye Akay. 2021. A discrete dynamic artificial bee colony with hyper-scout for RESTful web service API test suite generation. *Applied Soft Computing* 104 (2021), 107246.

[32] Abhinav Sharma, M Revathi, et al. 2018. Automated API testing. In *2018 3rd International Conference on Inventive Computation Technologies (ICICT)*. IEEE, 788–791.

[33] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. Resttestgen: automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 142–152.

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *CoRR* abs/2201.11903 (2022). arXiv:2201.11903 https://arxiv.org/abs/2201.11903

[35] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial testing of restful apis. In *Proceedings of the 44th International Conference on Software Engineering*. 426–437.

[36] Koji Yamamoto. 2021. Efficient penetration of API sequences to test stateful RESTful services. In *2021 IEEE International Conference on Web Services (ICWS)*. IEEE, 734–740.

[37] Man Zhang and Andrea Arcuri. 2021. Adaptive hypermutation for search-based system test generation: A study on rest apis with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–52.

[38] Man Zhang and Andrea Arcuri. 2021. Enhancing resource-based test case generation for RESTful APIs with SQL handling. In *International Symposium on Search Based Software Engineering*. Springer, 103–117.

[39] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based test case generation for restful web services. In *Proceedings of the genetic and evolutionary computation conference*. 1426–1434.

[40] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2021. Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering* 26, 4 (2021), 76.