SATORI: Static Test Oracle Generation for REST APIs

Juan C. Alonso
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
javalenzuela@us.es

Alberto Martin-Lopez
SEART @ Software Institute
Università della Svizzera italiana
Lugano, Switzerland
alberto.martin@usi.ch

Sergio Segura
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
sergiosegura@us.es

Gabriele Bavota
SEART @ Software Institute
Università della Svizzera italiana
Lugano, Switzerland
gabriele.bavota@usi.ch

Antonio Ruiz-Cortés
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
aruiz@us.es

***Abstract*—REST API test case generation tools are evolving rapidly, with growing capabilities for the automated generation of complex tests. However, despite their strengths in test data generation, these tools are constrained by the types of test oracles they support, often limited to crashes, regressions, and non-compliance with API specifications or design standards. This paper introduces SATORI (Static API Test ORacle Inference), a black-box approach for generating test oracles for REST APIs by analyzing their OpenAPI Specification. SATORI uses large language models to infer the expected behavior of an API by analyzing the properties of the response fields of its operations, such as their name and descriptions. To foster its adoption, we extended the PostmanAssertify tool to automatically convert the test oracles reported by SATORI into executable assertions. Evaluation results on 17 operations from 12 industrial APIs show that SATORI can automatically generate up to hundreds of valid test oracles per operation. SATORI achieved an F1-score of 74.3%, outperforming the state-of-the-art dynamic approach AGORA+ (69.3%)—which requires executing the API—when generating comparable oracle types. Moreover, our findings show that static and dynamic oracle inference methods are complementary: together, SATORI and AGORA+ found 90% of the oracles in our annotated ground-truth dataset. Notably, SATORI uncovered 18 bugs in popular APIs (Amadeus Hotel, Deutschebahn, FDIC, GitLab, Marvel, OMDb and Vimeo) leading to documentation updates by the API maintainers.**

***Index Terms*—REST APIs, test oracle, LLM, automated testing**

I. Introduction

Web Application Programming Interfaces (APIs) allow heterogeneous software systems to communicate over the network [1], [2]. Among these, REST APIs—those adhering to the REpresentational State Transfer (REST) architectural style [3]—have become the predominant standard. REST APIs organize their functionality around distinct resources (e.g., a video in the Vimeo API [4]) that clients access and manipulate through HTTP interactions. REST APIs underpin the business models of major companies such as Google, Microsoft, and Uber [1]. The Postman 2024 State of the API Report [5] shows that APIs are crucial business assets, with 62% of developers working on revenue-generating APIs.

The importance of REST APIs has led to the development of numerous techniques and tools for automated test case generation for these systems [6], [7]. Most techniques follow a black-box approach, deriving test cases automatically from the OpenAPI Specification (OAS) [8] of the API under test. These test cases are generated by assigning values to the input parameters and validating the returned responses using various test oracles [9], which serve as mechanisms for determining whether the output of a program is correct for a given input. Despite their promising results in generating valid API requests, these tools are all limited by the types of failures they can detect, primarily crashes (5XX HTTP status code responses) [10]–[16], disconformities with the API specification (e.g., an undocumented output JSON property) [11]–[15], regressions [17], and violations of API best practices (e.g., ensuring that repeated calls to idempotent operations return identical responses) [18]. For example, given the API specification of the "getBusinesses" operation of the Yelp API shown in Listing 1, Listing 2 shows an API response which conforms to such specification and would be considered correct by existing tools. However, this response may still contain errors that would go undetected by test case generators, such as incorrect field length (e.g., `country` should have 2 characters), format (e.g., `image_url` should be a valid URL), or violations of numerical constraints (e.g., `latitude` should range from -90 to 90), among others. Recent surveys [6] and tool comparisons [7], [19] highlight test oracle generation as a key challenge in automated test case creation for REST APIs. This is the problem that motivates our work.

To the best of our knowledge, the only existing approach in the literature that addresses the automated generation of test oracles for REST APIs is AGORA+ [20], [21], which generates test oracles through the detection of likely invariants (i.e., properties of the output that should always hold). Invari-

ants are detected by analyzing the API specification and a set of API requests with their corresponding responses. Although effective, the main limitation of AGORA+ is its reliance on a sufficiently diverse test suite that thoroughly exercises the API functionality to report accurate invariants. If the test suite lacks diversity or contains faulty responses, the reported invariants may be wrong or incomplete.

This paper presents SATORI (Static API Test ORacle Inference), a black-box static approach for automatically generating test oracles for REST APIs by analyzing their OAS document, without requiring prior API execution. SATORI leverages large language models (LLMs) to infer test oracles from the unstructured components of the OAS document, such as response field names and descriptions, making it compatible with existing API testing tools that support OAS. Currently, SATORI supports a catalog of 17 types of test oracles, which can be easily extended. To foster its adoption, we extended the PostmanAssertify tool [21] to transform the test oracles reported by SATORI into executable JavaScript assertions, written using the Chai library [22], that are compatible with Postman [23], a widely used API platform in industry with over 40 million users.

The results of an evaluation conducted on 17 operations from 12 industrial APIs show the capabilities of SATORI to automatically generate up to hundreds of valid test oracles per API operation, achieving an F1-Score of 74.3%, better than AGORA+ (69.3%) in generating the same types of oracles supported by both approaches. Moreover, SATORI identified 18 real bugs across 7 widely used industrial APIs (vs. 13 bugs in 7 APIs by AGORA+), which would have passed unnoticed by existing test case generators. Our findings led to documentation updates in the API of Vimeo. Since it does not require prior API execution, SATORI offers a more cost-effective solution than AGORA+. Our thorough evaluation also shows that each approach excels in identifying distinct oracle types, making them complementary: the combination of SATORI and AGORA+ found 90% of the test oracles of an annotated ground-truth dataset.

This paper makes the following research and engineering contributions:

- SATORI, a black-box static approach for automatically generating test oracles for REST APIs through specification analysis.
- OKAMI, a dataset containing the annotated ground truth of all the test oracles of the API operations used in our evaluation (over 10.5k test oracles from more than 1.8k response fields), enabling benchmarking and future comparisons. OKAMI is publicly available on Hugging Face [24].
- An extension of PostmanAssertify [21] that transforms the test oracles generated by SATORI into executable JavaScript assertions compatible with the widely used Postman API platform [23].
- An assessment of 21 LLMs as the backbone of SATORI, compared in terms of size, coding and reasoning capabilities, and cost.

```yaml
paths:
  '/businesses/search':
    get:
      operationId: getBusinesses
      parameters:
       - name: term
         description: 'Search term, e.g. food or restaurants.'
         in: query
         schema:
           type: string
       - name: location
         description: 'Geographic area for business search.'
         in: query
         schema:
           type: string
      responses:
        '200':
          description: 'Returns all businesses'
          content:
            application/json:
              schema:
                type: object
                properties:
                  total:
                    type: integer
                    description: 'Total number of businesses found.'
                  businesses:
                    type: array
                    items:
                      type: object
                      properties:
                        id:
                          type: string
                        name:
                          type: string
                        image_url:
                          type: string
                        rating:
                          type: number
                          description: 'Business rating (ranges from 1... 5).'
                        coordinates:
                          type: object
                          properties:
                            latitude:
                              type: number
                            longitude:
                              type: number
                        price:
                          type: string
                          description: 'Price level. Value is
                          one of $, $$, $$$ and $$$$.'
                          example: '$$'
                        location:
                          type: object
                          properties:
                            city:
                              type: string
                            country:
                              type: string
                              description: 'ISO 3166-1 alpha-2 country code.'
```

Listing 1: OAS excerpt of the Yelp API.

- An empirical evaluation of SATORI and AGORA+ in terms of precision, recall, F1-Score and failure detection across 17 operations from 12 industrial APIs, including the discovery of 22 real-world bugs.

All our code and data are publicly available [25].

II. Background and Related Work

A. Automated Testing of REST APIs

Web APIs commonly adhere to the REpresentational State Transfer (REST) [3] architectural style, being known as REST APIs [2]. REST APIs typically comprise multiple RESTful web services, each implementing CRUD (create, read, update, delete) operations on a resource (e.g., in the Vimeo API [4], a resource is a video). These operations are usually invoked by sending HTTP requests (generally `GET`, `POST`, `PUT` and `DELETE`) to a Uniform Resource Identifier (URI) representing a resource or a collection of resources.

REST APIs are commonly described using the OpenAPI Specification (OAS) [8] format, arguably the industry standard. An OAS document outlines the API operations, detailing their input parameters and responses. For instance, Listing 1 shows an excerpt from the OAS of the "getBusinesses" operation of the Yelp API [26]. The specification defines the HTTP method and URI required to call the operation (lines 1–3), operation

```
{
   "total": 1,
   "businesses": [
      {
         "id": "7dzGDH1BtzEjhZh1FeeaqA",
         "name": "Caipirinha Corner",
         "image_url": "https://s3-media1.fl.yelpcdn.com/bphoto/zrG.jpg",
         "rating": 4.0,
         "coordinates": {
            "latitude": 37.3968404980258,
            "longitude": -5.97877264022827
         },
         "price": "$",
         "location": {
            "city": "Seville",
            "country": "ES"
         }
      }
   ]
}
```


Listing 2: Yelp API response in JSON format.

ID (line 4), input parameters (lines 5–15), and response format (lines 16–60). Listing 2 shows an API response aligning with this specification.

Automated testing of REST APIs often employs a black-box approach [10], [13]–[18], [27]–[38], where, based on an OAS, these methods generate pseudo-random test cases (sequences of HTTP requests) and test oracles (assertions on the responses). Techniques vary in how they generate API calls (i.e., test inputs), leveraging methods like property-based testing [13], [15], [29], [39], model-based testing [27], [31], and constraint-based testing [12], [38], [40]. Some approaches target individual API operations and create single API requests, while others design sequences of API calls for stateful testing [10], [14], [15], [34], [37], [38]. White-box approaches, which require access to the API source code, are less common, and most existing techniques use search algorithms to maximize failure detection and code coverage [11], [41]. Recent approaches for API testing leverage LLMs [35], [38], [42], [43] and reinforcement learning [34], [37] to extract realistic input values and dependencies between parameters and operations, but none of them tackle the oracle problem.

Generated test oracles for failure detection primarily target API crashes (e.g., 5XX status codes) and API specification violations [11]–[15], with some also addressing regressions [17] and design practices [18]. However, these approaches are limited in identifying issues beyond syntax, overlooking domain-specific assertions like those in Listing 2. For example, they miss validations such as ensuring that the `country` response field value has two characters, `latitude` and `longitude` fall within specific ranges, or `price` adheres to allowed values (“$”, “$$”, “$$$”, “$$$$”).

Some approaches infer input or output constraints in REST APIs through static analysis [44], [45]. These constraints can be considered as test oracles in the form of pre- and post-conditions. However, these approaches require the source code of the system, which may not always be available (as in the case of the industrial APIs tested in our work) and thus cannot operate in black-box mode. More importantly, they derive constraints based on the *implemented* behavior, which may be faulty, thus limiting the usefulness of the inferred constraints.

To the best of our knowledge, the only approach for inferring domain-specific oracles for REST APIs is AGORA+ [20], [21], which uses invariant detection to generate test oracles. Invariants are output properties that should always hold (e.g., `LENGTH(return.location.country)==2`), and they are detected by analyzing patterns in previous API executions (i.e., request/response pairs). The effectiveness of AGORA+ depends on having a sufficiently diverse test suite: if the suite lacks variety or includes faulty responses, the detected invariants may be incomplete or invalid. However, many of these oracles can be inferred directly from the response field information in the OAS (e.g., response field names and descriptions, as shown in Listing 1), without prior API execution. This is the goal of SATORI.

B. Test Oracle Generation

Automated techniques for generating test oracles can be categorized by their inputs and the domains they target. Oracles can be derived from source code [46], [47], formal specifications [48], semi-structured documentation [49], previous program executions [50]–[54], or combinations of these inputs. Application contexts include databases [54], Java programs [50], cyber-physical systems [55], and machine learning programs [56], among others.

Other related techniques include metamorphic testing, regression testing and invariant detection. Metamorphic testing [29], [33], [39], [57] uses manually identified relationships between inputs and outputs across multiple executions of the system under test. Regression testing [58] compares the observed behavior to previous software versions to verify that changes do not disrupt existing features. Invariant detection identifies properties expected to consistently hold in program outputs, which can serve as test oracles to verify the correctness of outputs. They can be detected either statically, by analyzing code (without executing it) [59], or dynamically, by examining program behavior across executions [20], [21], [50], [60].

In recent years, LLMs have been applied across various stages of the software testing lifecycle [61], including unit test case generation [62]–[65], test input generation [66], [67], debugging [68], [69], and program repair [70]. LLM-based techniques proposed for tackling the oracle problem [47], [71]–[76] focus on specific programming languages and operate at the method level, leveraging information such as variable names and dataflow analysis to infer test oracles, thus making them unsuitable for the domain of REST APIs. To the best of our knowledge, SATORI is the first approach to leverage LLMs for addressing the oracle problem specifically in black-box testing of REST APIs.

III. SATORI

Figure 1 outlines SATORI, our approach for automatically generating test oracles for REST APIs through specification analysis. Starting from an OAS, SATORI extracts the schema of each response field of all the target operations and generates prompts for a (configurable) LLM. The outputs of the LLM (i.e., test oracles) are then processed into a machine-readable format and, following an optional human verification step, are provided to an extended version of PostmanAssertify [25] to produce a Postman collection with executable test assertions.

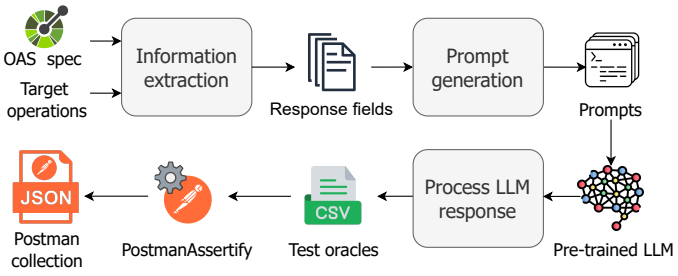


Fig. 1: Workflow of SATORI.

We now describe the complete SATORI workflow in detail, as well as the supported set of test oracles.

A. Automated Response Field Prompt Generation

This subsection explains how SATORI generates prompts to infer test oracles for each response field of an API operation.

1) Information Extraction: First, we extract contextual information to generate input prompts. Specifically, we gather details for each response field—such as name, description, and examples—along with global context like the API name and operation ID, providing the LLM with richer context to infer accurate test oracles.

2) Prompt Generation: Using information from the previous step, SATORI generates one prompt per response field. This prompt has been designed following well-established prompting techniques and patterns [77], [78]. We use a *System prompt* to set the overall behavior of the model and ensure consistent outputs. Then, the main prompt begins by establishing context for the task, followed by the necessary information, and then a detailed task description. Consequently, each prompt is structured into three sections: *Context prompt*, *Properties prompt*, and *Oracles prompt*. Our supplemental material [25] contains all the prompts generated by SATORI for our evaluation. In what follows, we provide an example of each prompt section, highlighting in boldface the dynamic parts of the prompt.

a) System Prompt: We use a role-playing approach, instructing the model to act as an expert software engineer.

> *You are a highly skilled software engineer with extensive experience in designing and testing REST APIs. Answer to your questions simply by generating a JSON object, without providing any additional information or explanation.*

b) Context Prompt: This provides the model with essential context, including the name of the API, the operation under test, and the name and type of the target response field.

> *I am going to give you a response field of the* ***getBusinesses*** *operation of the* ***Yelp*** *API. The name of this response field is "****price****" and it is of type* ***string****.*

c) Properties Prompt: This includes all additional properties of the response field available in the API specification, such as descriptions or examples, which the LLM can analyze to identify potential test oracles.

> *This response field has the following properties:*
> ***"name": "price"***
> ***"type": "string"***
> ***"description": "Price level. Value is one of $, $$, $$$, $$$$."***
> ***"example": "$$"***

d) Oracles Prompt: This final section guides the model to generate test oracles for the response field and is structured in three parts. First, the *Task introduction prompt* outlines the task. Next, several *Single oracle prompts* are presented as questions, guiding the model to infer specific test oracles based on the datatype of the response field (Section III-C). Each single oracle prompt consists of a question and specifies the expected response as a JSON property name and datatype, or a default JSON property value if no oracle is identified. Finally, the *Response format prompt* instructs the model to return the test oracles in a structured JSON.

> **Task introduction prompt**
> *Given this information, I want you to answer the following questions about some properties of this response field:*
> -
> **Single oracle prompts (one example)**
> ***3 - Should this response field have a set of specific values?*** *JSON property:*
> *"****string_specific_values****", of type* ***array of string****,* ***if there are no specific values, the array is empty***
> -
> **Response format prompt**
> *I want the response to be a single JSON object with the properties indicated in each question (****string_is_url, string_is_numeric, string_specific_values, string_is_email, string_is_date, string_fixed_length, string_is_time****). I don't want any kind of additional natural language explanation, only the JSON object.*

B. LLM Response Processing

SATORI processes the responses of the LLM to ensure syntactic correctness, handling issues like transforming responses into valid JSON, standardizing formats, merging multiple JSONs, and removing spurious text. Listing 3 shows the response returned by SATORI for the `price` response field.

```
{
  "string_is_url": false,
  "string_is_numeric": false,
  "string_specific_values": [ "$", "$$", "$$$", "$$$$" ],
  "string_is_email": false,
  "string_is_date": false,
  "string_fixed_length": null,
  "string_is_time": false
}
```

Listing 3: Example of test oracles generated by SATORI.

These test oracles, together with the OAS document, are provided as input to our PostmanAssertify extension, which transforms them into executable JavaScript assertions compatible with Postman [23]. PostmanAssertify produces a Postman collection of API requests for each tested operation, embedding the test oracles in each request and thus making the approach readily applicable in practice. For instance, the generated test cases could be executed programmatically, via the Postman GUI, or integrated into CI/CD pipelines. An example of a generated assertion is `pm.expect(["$", "$$", "$$$",`

"$$$$"].includes(price)).to.be.true, corresponding to the price response field (line 4 of Listing 3).

C. Target Oracles

SATORI supports a set of 17 types of test oracles, shown in Table I. Note that string, boolean and number oracles can be applied to elements of arrays (fourth row of Table I). These oracles support all 49 unary invariants (i.e., test oracles evaluating a single variable) supported by the dynamic approach AGORA+ [20], [21], which were derived from a systematic study of the oracles found in 40 real-world APIs. We focus on unary oracles to make our evaluation affordable, since deriving a ground-truth dataset of n-ary oracles would lead to a combinatorial explosion, requiring the manual annotation of a dataset of up to tens of thousands of instances per API operation (see Section IV-A1). Our proposed oracles are simpler than AGORA+'s while supporting the same use cases. For instance, float and integer invariants of AGORA+ (e.g., `OneOfFloat` and `OneOfScalar`) are combined into a single SATORI test oracle (e.g., `number_specific_values`). The resulting oracles assess properties such as string formats (e.g., `string_is_url`), numerical boundaries (e.g., `number_max_value`), and ordering of arrays (e.g., `array_number_asc_order`). We refer the reader to the SATORI documentation [25] for a complete list of the supported test oracles. These can be extended to support specific requirements.

IV. Evaluation

We aim to answer the following research questions:

RQ$_1$: *How do different LLMs perform in generating test oracles with SATORI?* We analyze the performance and cost of different LLMs as the backbone of SATORI, considering model size, code specialization and reasoning capabilities.

RQ$_2$: *What is the effectiveness of SATORI in generating test oracles and how does it compare against dynamic oracle generation approaches?* We compare the oracle generation capabilities of SATORI equipped with the LLM chosen in the previous RQ with respect to AGORA+ as a representative dynamic approach.

RQ$_3$: *How effective is SATORI in detecting artificially seeded faults and how does it compare against dynamic approaches?* We evaluate the effectiveness of the oracles generated by SATORI in detecting faults (mutations) in API responses, comparing it against AGORA+.

RQ$_4$: *How effective is SATORI in detecting real faults?* We evaluate the ability of SATORI to detect real faults in API responses, especially those not identified by AGORA+.

TABLE I: Test oracles supported by SATORI.

Datatype	Test oracles
String	`is_url`, `is_numeric`, `specific_values`, `is_email`, `is_date`, `fixed_length`, `is_time`
Boolean	`always_true`, `always_false`
Number	`min_value`, `max_value`, `specific_values`
Array	{String,Boolean,Number}-oracles, `min_size`, `max_size`, `specific_sizes`
Array[number]	{Array}-oracles, `asc_order`, `desc_order`

RQ$_5$: *How much does it cost to find a bug with SATORI? Can this cost be saved?* As GPT-4o is the most effective model with SATORI, we compute the cost per bug found (in dollars) and investigate whether free open-source models can find the same bugs.

A. Experiment 1: Test Oracle Generation

With this experiment, we aim to answer RQ$_1$ and RQ$_2$ by measuring the performance achieved by SATORI with different LLMs, and comparing it against dynamic oracle generation approaches.

1) Experimental Setup: Next we describe the dataset used as a benchmark, the LLMs and baselines experimented with, and the metrics considered for evaluation.

a) Dataset: As a contribution of this work, we present the OKAMI (Oracle Knowledge of API Methods for Innovation) dataset [24], a reliable benchmark for evaluating test oracle generation techniques for REST APIs. OKAMI is composed of 17 operations from 12 industrial APIs, which were used for the evaluation of the oracle generation approach AGORA+ [20], [21], [79] and in previous papers [10], [19], [28]. When necessary, we updated the OAS documents of these APIs according to the latest version of the web docs. We manually created the ground truth of all the oracles supported by SATORI for all the response fields of these API operations (e.g., labeling `href` response fields as URLs). Due to the extremely costly effort of manually annotating thousands of response fields, we randomly sampled API operations from the AGORA+ benchmark until having annotated, at least, 10k oracles. This resulted in a dataset of 17 API operations, 1,816 response fields and 10,645 test oracles. To avoid human bias or errors during the labeling process, we carefully analyzed the API specification (OAS) for labeling each response field and consulted the API providers in case of doubts or discrepancies. OKAMI is publicly available on Hugging Face [24] and as part of our supplemental material [25] to serve as a benchmark for future studies.

b) LLMs and Baselines: To answer RQ$_1$, we selected a set of 21 LLMs according to different criteria, namely: (i) model size (from 1B parameters to hundreds of billions of closed-source models), (ii) code specialization (explicitly trained on code or not), and (iii) reasoning capabilities (explicitly trained to reason about their answers or not). We selected models from six different vendors, i.e., Google, Meta, Microsoft, Alibaba, DeepSeek and OpenAI. We aim to analyze the impact of the aforementioned criteria on the performance of SATORI, and to select the best model for the subsequent experiments.

Regarding configuration parameters, we use the default settings for all models and a temperature of 0 (greedy decoding), thus making their outputs mostly deterministic.

To answer RQ$_2$, we compare the performance of SATORI against AGORA+, a dynamic approach that requires prior API execution to infer test oracles. In particular, we consider two versions of AGORA+: unary and binary. The unary version, denoted as AGORA+$_U$, reports the same test oracles

as SATORI (i.e., those involving a single variable), while the binary version, denoted as AGORA+, generates also test oracles involving two variables (e.g., `input.limit >= size(return.items[])`).

The authors of AGORA+ explain the need for a sufficiently diverse set of API requests and responses to detect invariants effectively, with 50 being enough. For a fair evaluation, we used the same sets of 10k requests used in the AGORA+ paper [21]. Since the performance of AGORA+ depends on these (randomly generated) sets of request-response pairs, the authors selected 10 subsets of 50 pairs each from among these 10k instances, and computed averages. We used the same 10 sets in this work.

c) Metrics: For both RQ_1 and RQ_2, we report the overall precision, recall and F1-Score. Here precision refers to the percentage of correct oracles generated by a technique T_i out of the total number of oracles generated by T_i. Recall, instead, captures the percentage of oracles in our ground truth dataset that has been generated by T_i. For RQ_2, we report also the same metrics per oracle type (see 17 types of oracles in Table I) for further analyses. For AGORA+ (binary), we report only the overall precision, since the OKAMI dataset contains only unary oracles. We also report the average time (in seconds) required to generate the test oracles for each response field of the API operations. The open-source LLMs were executed on a single NVIDIA A100 GPU with 80GB of VRAM, while OpenAI models were invoked via their web API [80]. AGORA+ does not require GPU resources, therefore it was executed on a desktop computer equipped with an Intel i9-12900K @3.20GHz, 64GB RAM, and 2TB SSD running Windows 11.

2) RQ_1: Experimental Results: Figure 2 shows the precision, recall and F1-Score achieved by each model. The figure is split in four subfigures according to the criteria previously mentioned, i.e., size (2a), code specialization (2b), reasoning capabilities (2c) and best model of each vendor considered (2d). The numbers shown on top of markers denote the average time (in seconds) to generate all possible oracles of a single response field in the OAS.

As expected, model size plays a role, as confirmed in Figure 2a. Here we evaluated four families featuring the same model in different sizes, namely, Phi-4 (3.8B, 14.7B), Gemma 3 (1B, 12.2B, 27.4B), Qwen2.5 and Qwen2.5-Coder (1.5B, 14.8B, 32.8B). As observed, models under 4B parameters exhibit significantly lower performance compared to the rest. However, models between 12-15B parameters achieve performance (69.3–71.6%) mostly on par with that achieved by models with ∼30B parameters (70.4–72.2%). This may be relevant if cutting costs is desirable (e.g., cheaper GPUs and faster inference times).

To evaluate the impact of code specialization on the task of oracle generation for REST APIs, we evaluated five models which offer a base version and a *code-specialized* version (i.e., the same model further trained on code), namely, Gemma 1.1 (8.5B), DeepSeek-V2-Lite (15.7B) and Qwen2.5 (1.5B, 14.8B, 32.8B). As shown in Figure 2b, results are mixed. While code specialization seems to help for Gemma, DeepSeek and Qwen 1.5B models (7.6% higher F1 on average), it has a slightly negative impact on Qwen 14.8B (-3.9% F1) and no significant impact on Qwen 32.8B. Execution times are not significantly affected by code specialization.

Regarding reasoning capabilities, we evaluated three models which offer versions distilled from (i.e., fine-tuned with the answers generated by) the reasoning model DeepSeek R1 [81], namely, Llama 3.1 (8B) and Qwen2.5 (14.8B, 32.8B). Figure 2c highlights two interesting aspects. First, reasoning distillation does not significantly affect the overall F1-Score of models, although it worsens precision and improves recall, meaning that distilled models tend to generate more oracles, resulting in more false positives (wrong oracles) but also less false negatives (less correct oracles missed). On the other hand, reasoning models are significantly slower than their non-distilled counterparts, taking 6-7× longer to generate oracles.

Figure 2d shows the comparison between the best model of each vendor and the closed source models GPT-4o and o3-mini. The average F1-Score ranges from 53.7% (DeepSeekCoder-V2-Lite) up to 74.3% (GPT-4o). The low performance of DeepSeekCoder-V2-Lite for this task may be attributed to the fact that is a *Mixture of Experts* (MOE) model originally designed with over 200B parameters, thus its lite version may not be able to unleash the full potential of the MOE architecture.

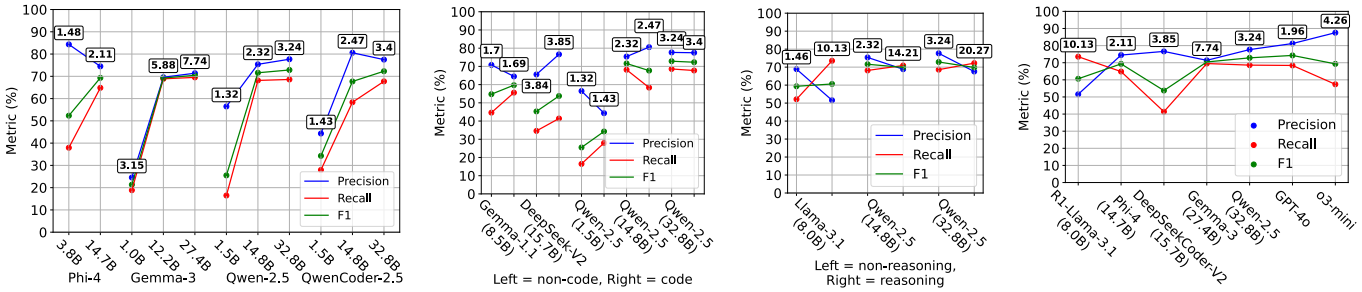
Answer to RQ_1: Comparison of LLMs

Models under 4B parameters exhibit significantly lower performance (<55% F1-Score) compared to their larger counterparts (∼70% F1-Score). Code specialization helps in some cases, while reasoning distillation does not affect the F1-Score, but increases execution times. Overall, the best model for SATORI is GPT-4o (74.3% F1-Score, 1.96s execution time).

3) RQ_2: Experimental Results: Table II shows the performance, in terms of precision (P), recall (R) and F1-Score (F1), as well as true and false positives and negatives (TP, TN, FP, FN) for each type of test oracle and overall achieved by SATORI (equipped with GPT-4o) and AGORA+$_U$. The table combines oracles related to primitive and array datatypes (e.g., `string_is_url` and `array_string_is_url` are both considered `string_is_url`). The table does not show oracles not found in the ground-truth dataset and for which no approach generated false positives (i.e., `array_number_desc_order`).

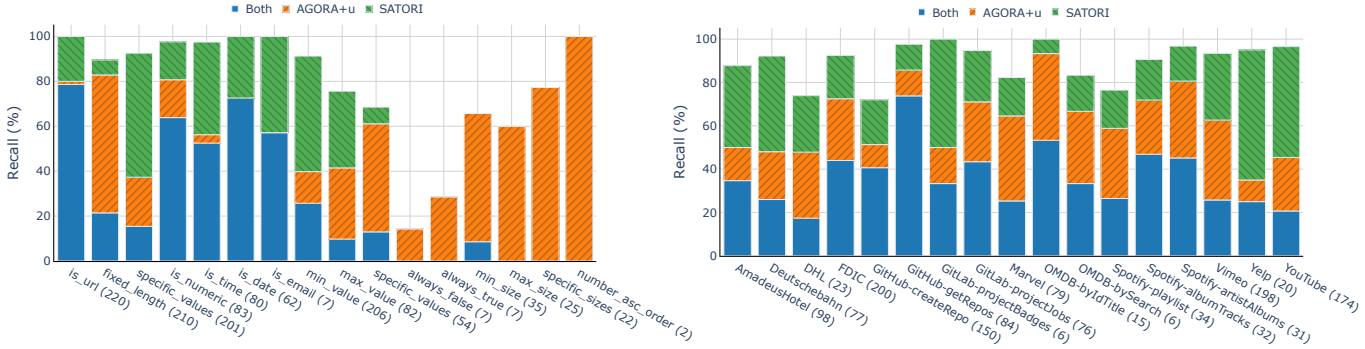
The results show that AGORA+$_U$ achieved higher F1-Score than SATORI for 9 out of 16 types of oracles, while SATORI is better for the remaining 7. Even so, the overall F1-Score of SATORI (74.3%) remains higher than that of AGORA+$_U$ (69.3%). For the binary version of AGORA+, the precision of AGORA+ is 68.8%, significantly lower than the precision of both SATORI (81.2%) and AGORA+$_U$ (76.8%), meaning that AGORA+ tends to generate significantly more false positives.

Figures 3a and 3b show the overlapping between SATORI



(a) Size. (b) Code specialization. (c) Reasoning capabilities. (d) Best model of each vendor.

Fig. 2: RQ_1: Precision, recall and F1-Score of each model evaluated according to several criteria. Numbers above markers denote average time (in seconds) to generate oracles for a single response field.



(a) Per oracle type. (b) Per API operation.

Fig. 3: RQ_2: Overlapping of recall (percentage of oracles detected) between SATORI and AGORA+$_U$.

TABLE II: RQ_2: Test oracle generation by SATORI and AGORA+$_U$, per oracle type and overall.

	Type & Oracle	AGORA+$_U$							SATORI (GPT-4o)						
		P	R	F1	TP	TN	FP	FN	P	R	F1	TP	TN	FP	FN
String	is_url	99.9	80	88.9	176.1	803.8	0.2	43.9	94.3	98.6	**96.4**	217	791	13	3
	fixed_length	81.9	83.6	**82.7**	173.9	777.4	38.6	34.1	59.6	28.6	38.7	59	778	40	147
	specific_values	47.6	42.5	44.9	74.5	766.6	82	100.9	65.1	85	**73.8**	142	781	76	25
	is_numeric	99.1	77.8	**87.2**	64.6	940.4	0.6	18.4	84.8	80.7	82.7	67	929	12	16
	is_time	100	55	70.9	44	944	0	36	97.4	93.8	**95.5**	75	942	2	5
	is_date	100	72.9	84.3	45.2	962	0	16.8	95.4	100	**97.6**	62	959	3	0
	is_email	100	42.9	59	3	1017	0	4	100	100	**100**	7	1017	0	0
Number	min_value	81.4	43.6	56.7	82.5	23.7	18.9	106.9	82	91.9	**86.6**	159	24	35	14
	max_value	66.2	44.2	52.9	34	137.5	17.5	43	90	44.4	**59.5**	36	147	4	45
	specific_values	65.6	66.3	**65.9**	33.7	163.4	17.8	17.1	78.6	20.4	32.4	11	175	3	43
Bool	always_false	10.5	14.3	**12**	1	117.3	8.7	6	-	0	-	0	126	0	7
	always_true	35.2	27.1	**30.3**	1.9	122.3	3.7	5.1	-	0	-	0	126	0	7
Array	min_size	58.4	63.5	**60.8**	22.3	64.9	16	12.8	100	8.6	15.8	3	81	0	32
	max_size	46.3	59.2	**51.9**	14.2	75.4	16.6	9.8	0	0	-	0	91	3	22
	specific_sizes	47.8	77.1	**59**	16.2	77.2	17.8	4.8	0	0	-	0	94	3	19
	number_asc_order	100	100	**100**	2	1	0	0	-	0	-	0	1	0	2
	TOTAL	**76.8**	**63.2**	**69.3**	**789.1**	**6993.9**	**238.4**	**459.6**	**81.2**	**68.4**	**74.3**	**838**	**7062**	**194**	**387**

and AGORA+$_U$ in terms of oracles detected (recall), grouped by oracle type and API operation, respectively. The numbers in parentheses in each tag indicate the number of possible oracles to be detected for each oracle type or API operation. Out of the 1167 oracles detected, 374 (32%) were identified exclusively by SATORI, 329 (28.2%) only by AGORA+$_U$, and 464 (39.8%) by both approaches. Looking at Figure 3a, we can see that both approaches achieve similar performance in certain oracles (e.g., `string_is_numeric` or `number_max_value`). In these cases, SATORI is more cost-effective, as it only requires access to the API specification, unlike AGORA+$_U$, which needs a diverse test suite to verify API behavior. However, when looking at the performance differences, it is clear that both approaches are complementary. For example, SATORI can precisely infer enum values from descriptions (`string_specific_values`) or detect domain-specific minimum values (e.g., -90 for `latitude`). The correct inference of these oracles for AGORA+$_U$ may be hard if the test suite used as input is not diverse enough (e.g., it does not include a request with a latitude value of -90). On the other hand, there are some types of oracles that only AGORA+$_U$ detected. This is explained by the fact that some oracles simply cannot be inferred from the specification, since they may not be explicitly stated in the OAS. Instead, they require the execution of the API to find such patterns, for instance, detecting a boolean field always being true (`boolean_always_true`) or the maximum size of an array (`array_max_size`).

When analyzing the performance of SATORI and AGORA+$_U$ per API operation (Figure 3b), the trend is similar to the one observed in the previous analysis. Although the overall recall of SATORI is higher than that of AGORA+$_U$, the latter detected more oracles in 10 out of 17 operations. This obviously comes at the cost of (i) executing the API, and (ii) being less precise, i.e., generating more false positives, as illustrated in Table II.

Answer to RQ_2: Static and dynamic test oracle generation effectiveness

Overall, SATORI outperforms AGORA+$_U$ in terms of precision, recall and F1-Score. However, their overlapping in terms of generated oracles shows that both approaches are complementary, and that SATORI can generate a significant percentage of the test oracles without previously executing the API.

B. Experiment 2: Artificial Fault Detection

This experiment aims to answer RQ_3 by comparing the effectiveness of the oracles generated by SATORI and AGORA+ in detecting failures caused by artificially seeded faults.

1) Experimental Setup: Next, we describe the setup for this experiment, detailing the techniques evaluated, the test oracles implemented, the test case selection criteria, the mutant generation process and the metrics used to measure performance.

a) Techniques: We evaluated SATORI, AGORA+, and their combination in detecting API failures caused by artificial faults. We distinguish between the results of AGORA+$_U$ and AGORA+ (i.e., leveraging also binary oracles). We also report the failures that could be detected with all unary oracles of the ground-truth dataset, i.e., an upper bound for both SATORI and AGORA+$_U$.

b) Test Oracles: For each API operation of Experiment 1, we selected the valid test oracles generated by each approach (i.e., confirmed as true positives) which were automatically transformed into executable assertions using PostmanAssertify [21].

c) Test Cases: For each API operation, we randomly selected 1k API requests and responses from the set of 10k previously used (see Section IV-A1) meeting the following constraints: (i) they were not part of the 50-request set used as input for AGORA+; (ii) they contained at least one result item (since we cannot apply mutation operators on empty arrays); and (iii) they revealed no failures (since mutation testing requires a green test suite).

d) Mutants: Since we do not have access to the source code of the APIs under test, we cannot apply traditional mutation testing techniques. Instead, we used a *black-box* approach by mutating directly the API responses. This is the same approach used by the authors of AGORA+ [20], [21] to evaluate the effectiveness of their approach.

We used JSONMutator [82] to introduce a *single error* in each API response, simulating a *failure* that could be caused by a *fault* in the API. JSONMutator is configured to apply mutation operators that result in syntactically valid mutants, i.e., conform to the API specification. Syntactically invalid mutants that would result in violations of the API specification (e.g., adding a new property to a JSON object) can be detected by existing approaches and therefore are out of the scope of both SATORI and AGORA+. Similarly, the mutation operators that convert response fields into null values are disabled, since null values can be easily detected as violations of the `nullable` property of OAS. The mutation operators applied in this context include modifications to boolean, number, and string values (e.g., by modifying or replacing values) as well as changes to array values (e.g., by removing elements or altering their order). In total, 12 different mutation operators are applied. All the mutations result in a distinguishable change in the API response and therefore there were no equivalent mutants [83]. Our supplementary material contains a detailed list of all the mutation operators applied [25].

TABLE III: RQ_3: # assertions (A) and % failure detection ratio (FDR) per API operation and overall by each approach.

	AGORA+$_U$		AGORA+ bin.		AGORA+		SATORI		Both	
API - Operation	**#A**	**FDR**	**#A**	**FDR**	**#A**	**FDR**	**#A**	**FDR**	**#A**	**FDR**
AmadeusHotel	47	56.4	22	3.8	69	60.2	70	48.5	107	66.9
Deutschebahn	32	19	6	0.8	38	19.8	53	16.8	73	26.5
DHL	10	45.6	3	2.8	13	48.3	10	34.8	19	51.2
FDIC	107	43.8	30	3.1	137	46.9	114	27.4	187	52.9
GitHub-createRepo	75	34.9	117	57.9	192	92.8	92	39.5	226	92.8
GitHub-getRepos	69	40.1	61	24.7	130	64.9	72	37.8	143	65.5
GitLab-getBadges	3	30.4	0	0	3	30.4	5	35.8	6	49.4
GitLab-projectJobs	42	25.5	11	11.6	53	37.2	48	23.9	83	39.8
Marvel	40	30.3	16	6.3	56	36.6	31	19	69	39.3
OMDB-byIdTitle	14	33.8	1	2.4	15	36.2	9	17.7	16	38
OMDB-bySearch	4	18.6	1	2	5	20.6	2	14.7	5	20.6
Spotify-playlist	18	46.7	22	46.1	40	92.8	15	28.9	46	92.8
Spotify-albumTracks	23	65.6	19	1.6	42	67.2	21	53.9	48	68.4
Spotify-artistAlbums	22	68.1	21	7.4	43	75.6	19	56.8	48	78.3
Vimeo	104	23.2	95	23.9	199	47.1	111	20.5	255	51.6
Yelp	7	11.5	5	12.1	12	23.6	17	15	24	31
YouTube	53	60.8	37	5.3	90	66.1	125	37.3	183	70.7
TOTAL	**670**	**38.5**	**467**	**12.5**	**1137**	**51**	**814**	**31.1**	**1538**	**55**

e) Metrics: For each mutated API response of the 1k test cases used, we ran the assertions and marked the failure as detected if at least one of the test assertions failed. Then, we computed the failure detection ratio (FDR) achieved by the approach on the test suite. We repeated the mutation process 100 times to minimize the effect of randomness, computing the average percentage of failures detected. In total, the results are based on 1.7M seeded errors: 17 operations $\times$ 1k API responses $\times$ 100 repetitions.

2) Experimental Results: Table III shows the number of assertions generated (i.e., true positive oracles) and the FDR achieved by SATORI, AGORA+ (unary, binary and combined), and the combination of both. SATORI achieved an average FDR of 31.1%, ranging from 14.7% to 56.8%. AGORA+$_U$ achieved an average FDR of 38.5%, ranging from 11.5% to 68.1%. The binary oracles of AGORA+ increased FDR an average of 12.5%, leading to an average FDR for AGORA+ of 51%. The combination of both SATORI and AGORA+ achieved a FDR of 55%, ranging from 20.6% to 92.8%. In terms of assertions, SATORI generated an average of 47.9 per API operation, more than AGORA+$_U$ (39.4) and less than AGORA+ (66.9).

While AGORA+$_U$ achieved a higher FDR than SATORI, two things are worth noting. First, SATORI uncovered 80.8% of the failures detected by AGORA+$_U$ (and 61% of the failures detected by AGORA+) without needing to execute the API, which represents a significant advantage in terms of cost-effectiveness. Second, SATORI managed to uncover new failures not detected by AGORA+$_U$, as shown in Figure 4, which represents the overlap (blue) of the FDR between SATORI (green) and AGORA+ (orange), as well as the FDR achieved

by the binary oracles (gray) and the optimal scenario of the ground-truth oracles (red). As illustrated, SATORI detected unique failures in 14 out of the 17 API operations. This means that SATORI can be used to complement AGORA+$_U$, as it can detect failures that AGORA+$_U$ cannot, and vice versa. The combination of both approaches achieved an FDR of 55%, which is significantly higher than the FDR of either approach alone.

Figure 4 also provides interesting insights regarding the strength of the unary oracles generated by SATORI and AGORA+$_U$ combined. In most APIs, the generated unary oracles (blue, green and orange bars) achieved an FDR very close to that achieved by the ground-truth oracles, i.e., the optimal scenario (red lines). In detail, the ground-truth oracles achieved an average FDR of 47.6% across all API operations. The automatically generated unary oracles achieved an average FDR of 44.3%, just 3.3% below the ground-truth oracles. This indicates that these unary oracles are very effective at detecting the failures that they are designed to detect. Intuitively, there are some failures that are impossible to detect even with the ground-truth oracles, such as subtle modifications to string fields which do not follow any format or the mutation of a number field within a certain valid range. Detecting such failures is extremely challenging and requires domain-specific knowledge or even manual inspection.

The APIs of GitHub, Spotify-playlist, Vimeo, and Yelp benefited from the binary test oracles of AGORA+ (gray bars), achieving a notable boost in FDR. This is primarily due to the presence of numerous equality (e.g., `input.description == return.description`), substring (e.g., `return.name substring of return.full_name`), and arithmetic (e.g., `return.total >= size(return.businesses[])`) comparisons in these APIs, which contribute to the inflated results.

The higher FDR (i.e., detecting more failures) of AGORA+ over SATORI (51% vs. 31.1%) does not necessarily mean that AGORA+ can catch more *real* bugs than SATORI. Our next experiment is designed to further explore this aspect.

Answer to RQ$_3$: Artificial fault detection capability

SATORI detected 61% of the failures detected by AGORA+ without previously executing the API under test, with an FDR ranging between 14.7% and 56.8%. More importantly, both approaches are complementary, achieving a combined FDR of 55%.

C. Experiment 3: Real Fault Detection

This experiment addresses RQ$_4$ by comparing the effectiveness of SATORI and AGORA+ in detecting failures caused by real faults.

1) Experimental Setup: We compared the performance of SATORI and AGORA+ in detecting real failures by converting the valid test oracles reported by each approach into executable assertions using PostmanAssertify. These assertions were then executed on the original dataset of 10k API requests from Experiment 1. Violations of these test oracles revealed real

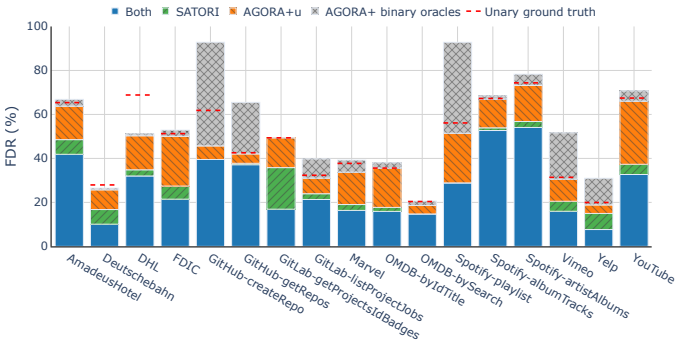


Fig. 4: FDR overlapping between SATORI and AGORA+.

TABLE IV: RQ$_4$: Faults detected by SATORI and AGORA+.

#Bug	Cat.	API-Operation	AGORA+	SATORI
1	1	AmadeusHotel	1	✓
2	2	Deutschebahn	10	✓
3	1	Deutschebahn	-	✓
4	1	Deutschebahn	-	✓
5	1	Deutschebahn	1	✓
6	2	FDIC	10	✓
7	2	FDIC	10	✓
8	2	FDIC	10	✓
9	2	FDIC	-	✓
10	2	FDIC	-	✓
11	2	FDIC	-	✓
12	2	FDIC	-	✓
13	4	GitHub-createRepo	9	-
14	1	GitLab-projectBadges	2	✓
15	2	Marvel	1	✓
16	3	Marvel	10	-
17	1	Marvel	10	-
18	1	Marvel	6	-
19	1	Marvel	-	✓
20	2	OMDB-bySearch	10	✓
21	2	Vimeo	-	✓
22	3	Vimeo	-	✓
Found always / Found at least once			7/13	**18/18**

bugs. For AGORA+, we repeated the experiment 10 times, each using a different random subset of 50 requests for invariant detection, and report the number of executions in which AGORA+ identified each bug.

2) Experimental Results: Table IV shows all bugs found among SATORI and AGORA+. SATORI found 18 bugs, while AGORA+ found 13 (although only 7 of them were consistently detected in all executions). In total, both approaches detected 22 unique bugs across 8 APIs; 13 of these bugs (bugs 1, 5, 6, 7, 8, 13, 14, 15, 16, 17, 18, 19, and 20) were also detected by the authors of AGORA+ [20], [21]; 2 of the new bugs found only by SATORI (bugs 21 and 22) have been confirmed by developers. Our supplementary material provides detailed bug descriptions, replication videos, and anonymized screenshots of reports and developer responses [25]. These bugs are grouped into the following four categories:

Category 1: Invalid String Formats. These bugs occur when a string response field is expected to follow a specific format (e.g., country codes, numbers, or timestamps), but the API returns values that deviate from this format. For example, in the "projectBadges" operation of the GitLab API (bug 14), SATORI and AGORA+ identified instances where the `rendered_image_url` response field, expected to be a URL, instead returned invalid URLs containing whitespaces.

Category 2: Invalid Enum Values. These bugs are found

when a response field is expected to hold specific string values (often outlined in the specification), but the API returns either undocumented values (bugs 9, 10, 11, 12, 15, 20, and 21) or entirely different ones (bugs 2, 6, 7, and 8). For instance, in the FDIC specification, the `CONSERVE` and `LAW_SASSER_FLG` fields are expected to use numerical flags ("1" or "0"), yet the API returns "Y" or "N" (bugs 6 and 7). Similarly, in the Vimeo API (bug 21), while the specification lists 14 valid values for the account field, the API also returns an undocumented value ("custom"). The Vimeo API providers confirmed this inconsistency and created an internal issue to fix it.

Category 3: Numerical Constraints. These bugs occur when a numerical response field does not comply with expected constraints, such as minimum or maximum values. For example, in the Vimeo API, the `field_of_view` response field should range from 30 to 90. However, SATORI detected three videos with values exceeding the upper limit (bug 22). This bug has been confirmed by the API providers, and they have created an internal issue to update the API documentation.

Category 4: Binary Test Oracles. These bugs occur when a test oracle involving two variables is violated, making them detectable only by AGORA+. The only bug of this category (bug 13) was found in the "GitHub-createRepo" operation, where the violation of the invariant `input.license_template==return.license.key` revealed 15 cases of repositories being created with incorrect licenses.

Answer to RQ_4: Real fault detection capability

SATORI effectively detected 18 real bugs in 7 APIs.

D. Cost-Effectiveness Analysis

Our last RQ investigates the monetary cost of using SATORI to automatically generate test oracles for REST APIs and find bugs in them.

One of the main advantages of SATORI is its cost-effectiveness. Unlike what intuitively might be expected, SATORI does not rely on LLMs to analyze API responses, but rather the API specification. This means that SATORI can be executed once for each API response field from which one would desire to extract test oracles, and then the generated oracles can be reused for all subsequent API calls. Following this approach, we computed the marginal inference cost of using SATORI with GPT-4o, the LLM that achieved the best performance in our experiments. As we considered 1,816 API response fields, we made 1,816 calls to the OpenAI API, which resulted in 716,529 input tokens and 101,949 output tokens. According to the OpenAI pricing at the time of performing the calls,[1] this resulted in a total of $5.11. As we found 18 bugs in total, the cost per bug is $0.28.

We also analyzed whether this cost could be reduced or avoided by using a free, open-source LLM executed locally. We selected the best-performing one from RQ_1, Qwen2.5-32B. The oracles generated by SATORI equipped with Qwen2.5-32B successfully found 17 bugs, missing only bug 10 in the FDIC API. This indicates that SATORI is highly effective even when relying on open-source, smaller language models. However, we note that using models like Qwen2.5-32B locally requires significant computational infrastructure (e.g., high-end GPUs), and thus their actual cost depends on the availability of such infrastructure and the volume of reuse.

[1] $5 per 1M input tokens ($2.5 if cached), $20 per 1M output tokens [84].

Answer to RQ_5: Cost-effectiveness analysis

In total, SATORI with GPT-4o found 18 bugs for $5.11 ($0.28 per bug). SATORI with Qwen2.5-32B could find 17 of these bugs.

V. Threats to Validity

We discuss the potential threats to the validity of our results, along with the actions taken to mitigate them.

Internal validity. *Are there factors that might affect the results of our evaluation?* For our experiments, we used the OAS documents provided in the AGORA+ supplementary material [79]. In all cases, we updated the OAS to reflect the latest version of the web documentation. It is possible that these specifications have errors and deviate from the documented API behavior. To mitigate this threat, the specification files were reviewed by at least two authors.

The effectiveness of AGORA+ largely depends on the diversity of the input test suite. For a fair evaluation, we followed the same approach by the authors and used their same test suites [79], leveraging the same 10 sets of 50 randomly generated request-response pairs and computing averages across the 10 runs. To address the potential variability of SATORI, we use the default settings for all models and a temperature of 0 (greedy decoding), making the outputs of the models mostly deterministic.

Manually creating the ground truth of test oracles for all the APIs may be affected by human biases or errors. To mitigate this, we carefully analyzed the API specification for each response field labeled and contacted the API providers in case of doubts or discrepancies. Our supplementary material contains evidence of the questions posed to API providers and their responses, as well as the full OKAMI dataset, which is publicly available for further review [25].

External validity. *To what extent can we generalize the findings of our investigation?* We evaluated SATORI using 21 different LLMs and a set of 17 operations from 12 APIs. Our conclusions may not fully generalize beyond this scope. To mitigate this threat, we selected LLMs of varying sizes and vendors, along with a set of widely-used APIs spanning diverse domains and sizes, and used in related studies.

The test oracles supported by SATORI may not generalize beyond the selected APIs. We minimized this threat by basing these oracles on the unary invariants supported by AGORA+, derived from a systematic analysis of 40 real-world APIs (702 operations) from diverse domains [28]. However, we note that this list of test oracles is not exhaustive, and SATORI is designed to be easily extended with additional oracles.

VI. Conclusions and Future Work

This paper introduces SATORI, a black-box static approach for generating test oracles for REST APIs from their specification using LLMs. SATORI analyzes the response fields of an API operation, providing them as inputs to a target LLM, which infers a set of pre-defined test oracles. SATORI then converts these inferred oracles into a machine-readable format compatible with an extended version of PostmanAssertify, a tool that transforms the oracles into executable Postman assertions. This integration makes SATORI readily applicable for practical use.

Evaluation results on a set of 17 operations from 12 industrial APIs show that SATORI can generate hundreds of valid test oracles per operation without executing the API. SATORI achieved an F1-Score of 74.3%. The differences in performance between SATORI and AGORA+ reveal complementary strengths, with each approach excelling at detecting distinct types of test oracles, and their combination achieving a failure detection ratio of 55%. SATORI identified 18 bugs across 7 widely used industrial APIs, leading to documentation updates in the API of Vimeo. Operating in black-box mode, SATORI can also be easily integrated with API testing tools that support OAS. As part of our future work, we intend to extend SATORI to support test oracles involving multiple variables.

References

[1] D. Jacobson, G. Brail, and D. Woods, *APIs: A Strategy Guide*, 2011.

[2] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*, 2013.

[3] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[4] "Vimeo REST API," 2025, accessed May 2025. [Online]. Available: https://developer.vimeo.com/api

[5] "Postman 2024 State of the API report," 2024, accessed May 2025. [Online]. Available: https://www.postman.com/state-of-api/2024/

[6] A. Golmohammadi, M. Zhang, and A. Arcuri, "Testing RESTful APIs: A Survey," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, Nov. 2023.

[7] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated Test Generation for REST APIs: No Time to Rest Yet," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022, 2022, p. 289–301.

[8] "OpenAPI Specification," 2025, accessed May 2025. [Online]. Available: https://www.openapis.org

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[10] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 748–758.

[11] A. Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, pp. 1–37, 2019.

[12] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs," in *International Conference on Service-Oriented Computing*, 2020, pp. 459–475.

[13] S. Karlsson, A. Causevic, and D. Sundmark, "QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs," in *International Conference on Software Testing, Validation and Verification*, 2020, pp. 131–141.

[14] E. Viglianisi, M. Dallago, and M. Ceccato, "RestTestGen: Automated Black-Box Testing of RESTful APIs," in *International Conference on Software Testing, Verification and Validation*, 2020.

[15] Z. Hatfield-Dodds and D. Dygalo, "Deriving Semantics-Aware Fuzzers from Web API Schemas," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 345–346.

[16] H. Wu, L. Xu, X. Niu, and C. Nie, "Combinatorial Testing of RESTful APIs," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, p. 426–437.

[17] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential Regression Testing for REST APIs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, 2020, p. 312–323.

[18] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking Security Properties of Cloud Services REST APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 387–397.

[19] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "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*, ser. ESEC/FSE 2022, 2022, p. 408–420.

[20] J. C. Alonso, S. Segura, and A. Ruiz-Cortés, "AGORA: Automated Generation of Test Oracles for REST APIs," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023, 2023, p. 1018–1030.

[21] J. C. Alonso, M. D. Ernst, S. Segura, and A. Ruiz-Cortés, "Test Oracle Generation for REST APIs," *ACM Trans. Softw. Eng. Methodol.*, Mar. 2025, just Accepted.

[22] "Chai Assertion Library." 2025, accessed May 2025. [Online]. Available: https://www.chaijs.com/api/bdd/

[23] "Postman API Platform," 2025, accessed May 2025. [Online]. Available: https://www.postman.com

[24] "OKAMI dataset," 2025, accessed August 2025. [Online]. Available: https://huggingface.co/datasets/javalenzuela/okami-dataset

[25] "SATORI Replication package," 2025, accessed May 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15529767

[26] "Yelp API," 2025, accessed May 2025. [Online]. Available: https://docs.developer.yelp.com/docs/getting-started

[27] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Automated Black-Box Testing of RESTful Web APIs," in *International Symposium on Software Testing and Analysis*, 2021.

[28] J. C. Alonso, A. Martin-Lopez, S. Segura, J. M. Garcia, and A. Ruiz-Cortes, "ARTE: Automated Generation of Realistic Test Inputs for Web APIs," *IEEE Transactions on Software Engineering*, 2022.

[29] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic Testing of RESTful Web APIs," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2018.

[30] P. Godefroid, B.-Y. Huang, and M. Polishchuk, "Intelligent REST API Data Fuzzing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 725–736.

[31] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao, "Morest: Model-Based RESTful API Testing with Execution Feedback," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, p. 1406–1417.

[32] M. Kim, D. Corradini, S. Sinha, A. Orso, M. Pasqua, R. Tzoref-Brill, and M. Ceccato, "Enhancing REST API Testing with NLP Techniques," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1232–1243.

[33] L. Pan, S. Cohney, T. Murray, and V.-T. Pham, "EDEFuzz: A Web API Fuzzer for Excessive Data Exposures," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.

[34] M. Kim, S. Sinha, and A. Orso, "Adaptive REST API Testing with Reinforcement Learning," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 446–458.

[35] M. Kim, T. Stennett, D. Shah, S. Sinha, and A. Orso, "Leveraging Large Language Models to Improve REST API Testing," 2024.

[36] R. Yandrapally, S. Sinha, R. Tzoref-Brill, and A. Mesbah, "Carving UI Tests to Generate API Tests and API Specification," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1971–1982.

[37] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato, "DeepREST: Automated Test Case Generation for REST APIs Exploiting Deep Reinforcement Learning," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24, 2024, p. 1383–1394.

[38] T. Le, T. Tran, D. Cao, V. Le, T. N. Nguyen, and V. Nguyen, " KAT: Dependency-Aware Automated API Testing with Large Language Models ," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, May 2024, pp. 82–92.

[39] S. Segura, J. C. Alonso, A. Martin-Lopez, A. Durán, J. Troya, and A. Ruiz-Cortés, "Automated generation of metamorphic relations for query-based systems," in *2022 IEEE/ACM 7th International Workshop on Metamorphic Testing (MET)*, 2022, pp. 48–55.

[40] A. Martin-Lopez, S. Segura, C. Müller, and A. Ruiz-Cortés, "Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs," *IEEE Transactions on Services Computing*, 2021.

[41] D. Stallenberg, M. Olsthoorn, and A. Panichella, "Improving Test Case Generation for REST APIs through Hierarchical Clustering," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21, 2022, p. 117–128.

[42] M. Kim, T. Stennett, S. Sinha, and A. Orso, "A Multi-Agent Approach for REST API Testing with Semantic Graphs and LLM-Driven Inputs," 2024.

[43] M. Kim, S. Sinha, and A. Orso, "LlamaRestTest: Effective REST API Testing with Small Language Models," 2025.

[44] H. Grent, A. Akimov, and M. Aniche, "Automatically identifying parameter constraints in complex web APIs: a case study at Adyen," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 71–80.

[45] R. Huang, M. Motwani, I. Martinez, and A. Orso, "Generating REST API specifications through static analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[46] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21, 2021, p. 336–347.

[47] H. Yu, Y. Lou, K. Sun, D. Ran, T. Xie, D. Hao, Y. Li, G. Li, and Q. Wang, "Automated Assertion Generation via Information Retrieval and Its Integration with Deep Learning," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, p. 163–174.

[48] G. Gay, S. Rayadurgam, and M. P. Heimdahl, "Improving the Accuracy of Oracle Verdicts through Automated Model Steering," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014, p. 527–538.

[49] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, "C2S: Translating Natural Language Comments to Formal Program Specifications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, New York, NY, USA, 2020, p. 25–37.

[50] F. Molina, M. d'Amorim, and N. Aguirre, "Fuzzing class specifications," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, New York, NY, USA, 2022, p. 1008–1020.

[51] T. Chen, K. Heo, and M. Raghothaman, "Boosting static analysis accuracy with instrumented test executions," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, 2021, p. 1154–1165.

[52] C. G. Kapugama, V.-T. Pham, A. Aleti, and M. Böhme, "Human-in-the-loop oracle learning for semantic bugs in string processing programs," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022, 2022, p. 215–226.

[53] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, "Perfect is the enemy of test oracle," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 70–81.

[54] W. Dou, Z. Cui, Q. Dai, J. Song, D. Wang, Y. Gao, W. Wang, J. Wei, L. Chen, H. Wang, H. Zhong, and T. Huang, "Detecting isolation bugs via transaction oracle construction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1123–1135.

[55] J. Ayerdi, V. Terragni, A. Arrieta, P. Tonella, G. Sagardui, and M. Arratibel, "Generating metamorphic relations for cyber-physical systems with genetic programming: An industrial case study," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, 2021, p. 1264–1274.

[56] H. B. Braiek and F. Khomh, "On testing machine learning programs," *Journal of Systems and Software*, vol. 164, p. 110542, 2020.

[57] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[58] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[59] P. Cousot and R. Cousot, "Abstract interpretation frameworks," *Journal of Logic and Computation*, vol. 2, 08 1992.

[60] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007, special issue on Experimental Software and Toolkits.

[61] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, p. 911–936, Feb. 2024.

[62] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024.

[63] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and Improving ChatGPT for Unit Test Generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024.

[64] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 185–196.

[65] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit Test Generation using Generative AI: A Comparative Performance Analysis of Autogeneration Tools," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, ser. LLM4Code '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 54–61.

[66] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4All: Universal Fuzzing with Large Language Models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.

[67] M. Sun, Y. Yang, Y. Wang, M. Wen, H. Jia, and Y. Zhou, "SMT Solver Validation Empowered by Large Pre-Trained Language Models," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 1288–1300.

[68] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.

[69] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S.-C. Cheung, and J. Kramer, "Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 14–26.

[70] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What makes good in-context demonstrations for code intelligence tasks with llms?" in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23, 2024, p. 761–773.

[71] S. B. Hossain and M. Dwyer, "Togll: Correct and strong test oracle generation with llms," 2024.

[72] S. B. Hossain, A. Filieri, M. B. Dwyer, S. Elbaum, and W. Visser, "Neural-based test oracle generation: A large-scale evaluation and lessons learned," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 120–132.

[73] Y. He, J. Huang, H. Yu, and T. Xie, "An empirical study on focal methods in deep-learning-based approaches for assertion generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024.
[74] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, "Can large language models transform natural language intent into formal method postconditions?" *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024.
[75] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, ser. AST '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 54–64.
[76] D. Molinelli, A. Martin-Lopez, E. Zackrone, B. Eken, M. D. Ernst, and M. Pezzè, "Tratto: A neuro-symbolic approach to deriving axiomatic test oracles," vol. 2, no. ISSTA. New York, NY, USA: Association for Computing Machinery, Jun. 2025.
[77] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," in *Proceedings of the 30th Conference on Pattern Languages of Programs*, ser. PLoP '23, 2023.
[78] S. Schulhoff, M. Ilie, N. Balepur, K. Kahadze, A. Liu, C. Si, Y. Li, A. Gupta, H. Han, S. Schulhoff, P. S. Dulepet, S. Vidyadhara, D. Ki, S. Agrawal, C. Pham, G. Kroiz, F. Li, H. Tao, A. Srivastava, H. D. Costa, S. Gupta, M. L. Rogers, I. Goncearenco, G. Sarli, I. Galynker, D. Peskoff, M. Carpuat, J. White, S. Anadkat, A. Hoyle, and P. Resnik, "The prompt report: A systematic survey of prompt engineering techniques," 2025.
[79] "AGORA+ Replication package," 2025, accessed May 2025. [Online]. Available: https://doi.org/10.5281/zenodo.12506791
[80] "OpenAI API," 2025, accessed May 2025. [Online]. Available: https://platform.openai.com/docs/overview
[81] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," 2025.
[82] "JSONMutator," 2025, accessed May 2025. [Online]. Available: https://github.com/isa-group/JSONmutator
[83] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," ser. Advances in Computers, 2019, vol. 112, pp. 275–378.
[84] "OpenAI API Pricing," 2025, accessed May 2025. [Online]. Available: https://openai.com/api/pricing/