Analyzing Status Code Misuses in REST API Specifications

Alix Decrop[1], Mike Papadakis[2], and Gilles Perrouin[1]

[1] NADI, University of Namur, Namur, Belgium
{alix.decrop,gilles.perrouin}@unamur.be
[2] SnT, University of Luxembourg, Luxembourg, Luxembourg
michail.papadakis@uni.lu

Abstract. Most web APIs now rely on the REST architectural style, exploiting HTTP for client-server interactions. Clients typically interpret server responses using status codes, such as `200 OK` for a success or `404 Not Found` for an unavailable resource. However, as REST principles do not strictly enforce HTTP standards, servers may misuse status codes in their responses. For instance, a server might include a body with `204 No Content` or use `500 Internal Server Error` for a client error instead of using a `4xx` code. Such misuses cause inaccuracies in testing tools and confuse clients relying on status codes. To address this problem, we present SCOAS, a tool designed to detect and report status code misuses in REST API specifications. SCOAS defines 24 rules derived from HTTP standards, verifying if status codes are used correctly. Our evaluation demonstrates that status code misuses are systematic in REST APIs, with a total of 17,767 rule violations detected across 60 OpenAPI specifications. We also discuss implications and provide recommendations for APIs practitioners.

Keywords: REST API · OpenAPI Specification · Status Code · Misuse

1 Introduction

Many modern web Application Programming Interfaces (APIs) rely on the REpresentational State Transfer (REST) architectural style [12]. Such REST (also termed RESTful) APIs handle client-server communications through HTTP. Via its comprehensive list of methods and status codes, the HTTP standard allows for interoperable communication among web services. However, this paper finds that many REST APIs deviate from HTTP semantics regarding the use of status codes; We call such deviations *status codes misuses*.

Status code misuses can have several harmful consequences. For instance, some APIs returns status codes in the `5xx` range for client errors. However, when a client error is encountered, a code in the `4xx` range is expected, as `5xx` codes are exclusively reserved for server errors. This misleads clients in considering client-side errors as server-side. Misuses also deceive REST API analysis and testing tools, causing result inaccuracies (e.g., fuzzing tools often treat `5xx` codes as successful bug findings). Additionally, APIs may implement their own status codes

in their response bodies, forcing clients to implement non-interoperable parsing to inform users. APIs may also return overly generic status codes, such as `400 Bad Request` for an unsupported content format instead of `415 Unsupported Media Type`. These misuses are detrimental to API users, as debugging invalid requests become a tedious task due to a lack of suitable information.

Thus, this paper investigates HTTP status code misuses in REST APIs, whose prevalence has not been explored so far. In particular, this paper presents the following contributions: **(1)** SCOAS (Status Code Analysis in OpenAPI Specifications), a tool aimed at detecting status code misuses in REST API specifications. **(2)** A set of 24 status code usage rules, which we derived from HTTP standards and REST API usage. **(3)** A study of status codes misuses in 60 REST API specifications. A total of 17,767 misuses were found, also occurring in "popular" APIs (GitHub, Spotify, Stripe, etc.). **(4)** A replication package containing our implementation and evaluation data [7].

2 Background

REST APIs and HTTP. REpresentational State Transfer (REST) [12] is an architectural style providing a set of design principles for building scalable, loosely coupled, and interoperable web services. Such principles include stateless communication, where client requests must contain all necessary information for server processing. REST Application Programming Interfaces (APIs) use the HyperText Transfer Protocol (HTTP) [11] to perform CRUD (Create, Read, Update, Delete) operations on data resources identified by URIs.

HTTP Requests. To communicate with API servers, clients send HTTP requests containing a method (e.g., `GET`), path (e.g., `/users`), and optionally query parameters (e.g., `name=john`) or headers (e.g., `Accept: text/html`). HTTP implements nine methods to indicate the purpose of requests, with five of them frequently used: `GET` (request data), `POST` (send/create data), `PUT` (create/update data), `PATCH` (update data), and `DELETE` (delete data). With `POST`, `PUT`, and `PATCH`, an accompanying data payload is also sent to the server. For instance, a client could send the request `POST /users` with a JSON payload `{"name": "John"}` to create a user named John on the server. Later, the client could send the request `GET /users?name=John` (or `GET /users/John`, depending on the API implementation) to retrieve the data of the newly created user.

HTTP Status Codes. HTTP also defines a set of status codes to communicate the outcome of client-server interactions. Status codes are grouped into five ranges: `1xx` (informational), `2xx` (success), `3xx` (redirection), `4xx` (client error), and `5xx` (server error). A status code is structured as a three-digit identifier, where the first digit indicates the range and the next two digits indicate the exact condition within that range. For instance, the status code `200 OK` indicates a successful request, `404 Not Found` indicates that a resource was not found, and `500 Internal Server Error` indicates a server-side failure. Consequently,

status codes provide clear feedback to clients, facilitating error handling and debugging practices if used correctly.

OpenAPI Specification. The OpenAPI Specification (OAS) [26], previously known as Swagger, is a widely adopted industry standard to document REST APIs. The OAS standard is structured and machine-readable, written with either JSON or YAML file formats. Specifications can be organized into objects that describe the overall API (e.g., title, version, servers), resources and their paths, supported HTTP methods, request parameters, expected responses, and reusable components such as security schemes and data models. OAS is widely used for REST API testing practices, as witnessed in state-of-the-art testing tools [18,5,15,20,27,10] requiring OAS files as input. Moreover, editing tools such as the Swagger Editor [25] can convert OAS files into human-readable documents.

Related Work. Regarding the HTTP specification, Rautenstrauch et al. [23] analyzed the potential security impacts of violating HTTP specification rules. Benhabbour et al. [3] verified if network middleboxes affect the conformance of HTTP traffic. In terms of REST API design, Di Meglio et al. [9] analyzed RESTful design rule violations in web apps built by students. Bogner et al. [4] implemented RESTRuler, a tool aimed at detecting design rule violations in OpenAPI descriptions. Rodriguez et al. [24] did a large scale analysis of compliance with REST API principles and best practices. However, to the best of our knowledge, our work is the first of its kind to analyze status code misuses in REST API specifications.

Terminology. In this paper, we use the term API as a shorthand for a REST API. When we mention a REST API specification, this refers to a documentation written in the OpenAPI Specification (OAS) standard. We also interchangeably use the terms "status code misuse" and "status code usage rule violation".

3 Approach

To detect status code misuses, we developed SCOAS (Status Code Analysis in OpenAPI Specifications). Our tool is implemented in Python, and only requires one extra package (Jinja2, to dynamically render the HTML reports). Figure 1 illustrates a schema of our tool, with orange circles (X) representing the various phases. First, an OpenAPI Specification (A) of a REST API is provided by the user as input, in a JSON file. To analyze the specification, we begin by iterating over all described paths (B). For each path, we iterate over all described methods (C). For each method in each path, we iterate over all described responses (D). Doing so allows us to generate a status code context (E), composed of a path, method, and response. This is especially useful, as some status code usage rules rely on path data (e.g., path parameters such as `/{id}` for `404 Not Found` responses) and/or method data (e.g., `GET` methods require `200 OK` responses). When a status code context is found, we iterate over all defined status code usage rules (F) (defined in Section 4.2). We verify if one or more rules are violated

(i.e., if there are status code misuses for the context). If so, we generate a rule violation (misuse) context G containing the triggering context and information about the misuse. We append it to a list of violations H for output purposes. We continue the process until all status code contexts have been analyzed. When the iterations are completed (B - E), the tool terminates. An HTML report I and a file containing the execution logs J are generated as outputs.

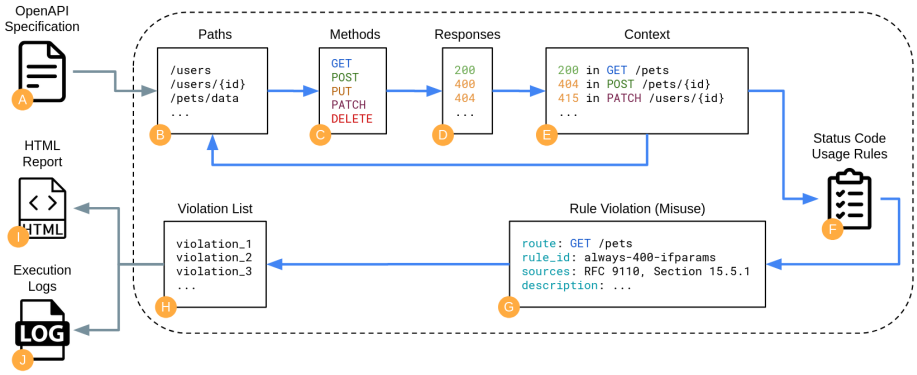

Fig. 1. SCOAS Overview.

4 Evaluation

Research Questions. For our evaluation, we formulate the following:

- **RQ.1** *Which status codes are used in REST APIs?* First, we aim to identify the distribution of status codes that are used in REST APIs responses. Doing so allows us to identify status code patterns related to REST API design, which can be used as a baseline to define rules in RQ.2.
- **RQ.2** *What status code usage rules can be defined for REST APIs?* Next, we aim to define status code usage rules based on the analysis of HTTP standards and results obtained from RQ.1. Rules violations consist in status code misuses.
- **RQ.3** *What is the prevalence of status code misuses in REST API specifications?* Finally, we aim to analyze and discuss the prevalence of status code misuses in real-world REST API specifications using SCOAS.

Benchmark. We formed a comprehensive benchmark containing 60 unique specifications (in the OAS format) from various REST APIs. These specifications were extracted from the Public REST API Benchmark (PRAB) [8], a state-of-the-art and peer reviewed benchmark in the REST API literature. The complete list of APIs is provided in our replication package [7].

Experimental Setup. Our evaluation was conducted using a laptop with a 2.4GHz processor and 16GB of RAM. SCOAS is fully deterministic and does not require a network connection to perform analyses. Hence, we performed a single execution of the tool for validation. On average, SCOAS took 5 seconds to process a specification.

4.1 RQ.1 - Status Codes in REST APIs

Before establishing status code usage rules, we first need to examine which status codes are employed in REST API responses. Understanding this distribution allows us to highlight the status codes that are most related to REST API design and usage. To this end, we analyzed the specifications included in our benchmark dataset. For each specification, we iterated over all defined paths (e.g., `/pets`), then examined all HTTP methods associated with each path (e.g., `PATCH`). For every method, we further inspected the corresponding response definitions to extract the status codes (e.g., `201 Created`). This procedure enabled us to quantify the total number of occurrences for each status code across the benchmark. Figure 2 presents the distribution of status codes per REST API.

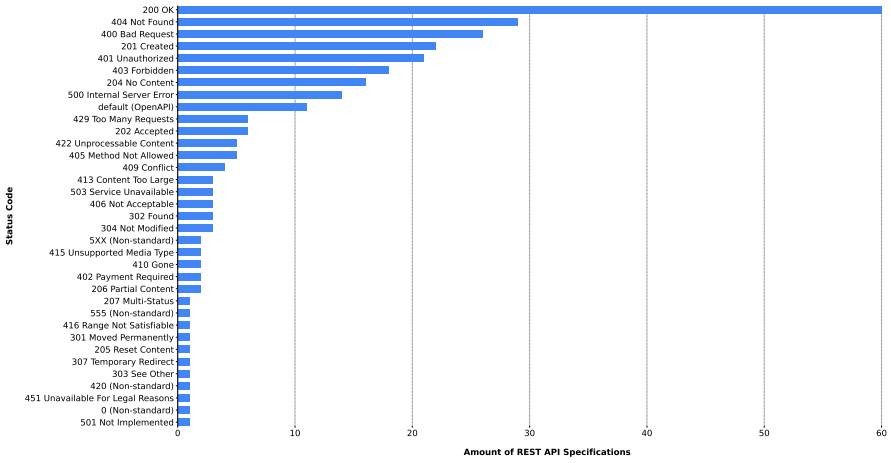

Fig. 2. Distribution of HTTP status codes across individual REST API specifications of the benchmark.

In total, we identified 35 distinct HTTP status codes, accounting for 12,028 occurrences. Among these status codes, `0`, `420`, `555`, `5XX`, and `default` are not defined in the HTTP standard (`5xx` is a range and not a code in itself). The `default` status code is defined in the OpenAPI Specification standard, and is used to collectively describe errors that have the same response structure. While this status code is non-standard, it is widely adopted in OAS files and can be

considered as valid when documenting REST APIs through this format. Our analysis led to the following observations.

The `200 OK` status code is the most frequently used, appearing 3,358 times and present in all 60 REST API specifications. This aligns with its expected role of indicating successful requests. However, as detailed in Section 4.3, the `200 OK` status code is sometimes used by default instead of employing more specific codes. Consequently, while widely used for its intended purpose, this status code is also prone to misuse. We also observe that the `201 Created` and `204 No Content` status codes are frequently used, aligning with their use for creation and deletion operations in REST APIs. Other status codes from the `2xx` (success) range appear less, as they are less used for typical CRUD operations.

The `404 Not Found` status code is prevalent, consistent with its function as a common client error code signaling that a requested resource is unavailable.

Several non-standard status codes were identified, including `0`, `420`, `555`, and `5XX`. Since OAS does not enforce status code validity, such codes may be used to represent API-specific behaviors. This highlights the importance of adhering to standard status codes, as external clients cannot reliably interpret the meaning of non-standard codes in responses.

No status codes were observed in the `1xx` (informational) range. Indeed, such status codes are primarily used for protocol-level signaling rather than final responses, thus untypical for REST interactions.

Similarly, there is a lack of status codes from the `3xx` (redirection) range. This is because REST API clients generally expect a direct response rather than being redirected (e.g., a simple `GET` request to retrieve data directly).

Lastly, we observe that the status codes `401 Unauthorized` (i.e., the client lacks authentication credentials) and `403 Forbidden` (i.e., the client is not authorized to access the resource) appear in 21 and 18 distinct specifications respectively. This suggests that several REST APIs employ frequent access control verification and implement authentication mechanisms for various purposes (e.g., pricing, rate limiting, admin paths, etc.).

RQ.1 Summary: After analyzing 60 REST API specifications, we identified 35 distinct status codes (five being non-standard), totaling 12,028 occurrences. The most frequent are `200` and `404`. There were no/few codes in the `1xx` and `3xx` ranges (less used in typical REST interactions). `401` and `403` also occurred, highlighting authentication mechanisms.

4.2 RQ.2 - Status Code Usage Rules for REST APIs

To define a set of status code usage rules, we read and analyzed the "Hypertext Transfer Protocol (HTTP) Status Code Registry" provided by the Internet Assigned Numbers Authority (IANA) [16]. It serves as the official registry for status codes. The registry details all status codes along with their official RFC (Request For Comments) Internet Standards, describing each status code with additional information and details. Most status code references redirect to RFC9110 [11]

Table 1. Selected HTTP status codes based on typical REST API interactions.

Status Code	Description
`200 OK`	Standard success.
`201 Created`	Data creation success.
`204 No Content`	Success without content.
`400 Bad Request`	Malformed request syntax.
`401 Unauthorized`	Insufficient authentication.
`403 Forbidden`	Insufficient permissions.
`404 Not Found`	Unavailable resource.
`406 Not Acceptable`	Unsupported `Accept` header.
`413 Content Too Large`	Unsupported `Content-Length` header.
`415 Unsupported Media Type`	Unsupported `Content-Type` header.
`422 Unprocessable Content`	Malformed request semantics.

(entitled "HTTP Semantics"), which is the latest and most up-to-date RFC for status codes as of 2026.

Status Code Subset. As not all status codes are suitable for REST interactions (i.e., CRUD operations), we identified a relevant subset for such purpose. To do so, we considered the `2xx` (successful) and `4xx` (client error) ranges, as they are sufficient for typical client-server interactions in REST APIs. Indeed, they can be used to inform clients that either their request has succeeded or their request has failed due to a client-side error. We did not consider the `1xx` (informational) and `3xx` (redirection) ranges, as RQ.1 demonstrated that these ranges are very rare and unusual in REST interactions. We also excluded the `5xx` (server error) range, as it describes errors unrelated to the interaction itself (e.g., server maintenance, bad gateway, unsupported HTTP, etc.). Table 1 presents each **Status Code** that was selected along with its corresponding **Description**. We covered three status codes for successes (`200`, `201`, `204`), and eight status codes for client errors (`400`, `401`, `403`, `404`, `406`, `413`, `415`, `422`). These status codes can handle a wide range of typical client-server interactions; For instance, if a client sends the request `GET /pets/romeo` and the name `romeo` does not exist, the server can respond with `404 Not Found`. If a client specifies `Accept: application/rtf` but only `application/json` is supported, the server can respond with `406 Not Acceptable`, and so on.

Status Code Usage Rules. Based on the relevant status codes selected, we were able to define a total of 24 unique status code usage rules. Table 2 presents the rules, which are represented by a unique **Identifier** and a **Description**. The usage rules can be categorized into two distinct groups: rules that verify if a status code is implemented (`has`), and rules that verify if a non-applicable status code is not implemented (`no`), based on a given context (cf. Section 3).

> **RQ.2 Summary**: With a subset of status codes suitable for REST interactions, we defined a total of 24 status code usage rules. They can be used to find two types of status code misuses: missing status codes that should be mandatory, and implemented status codes that are not applicable to a given context.

Table 2. Status code usage rules. **I** = Implement, **NI** = Never Implement.

Identifier	Description
200-if-get	**I** 200 OK in a GET method.
200-or-201-or-204-if-post	**I** 200 OK, 201 Created, or 204 No Content in a POST method.
200-or-201-or-204-if-put	**I** 200 OK, 201 Created, or 204 No Content in a PUT method.
200-or-204-if-delete	**I** 200 OK or 204 No Content in a DELETE method.
200-or-204-if-patch	**I** 200 OK or 204 No Content in a PATCH method.
204-if-no-content	**I** 204 No Content for a response that does not have content.
400-if-params	**I** 400 Bad Request if there are parameters (syntax).
400-if-payload	**I** 400 Bad Request if there is a payload (syntax).
404-if-path	**I** 404 Not Found if there are path parameters.
406-if-accept	**I** 406 Not Acceptable for unsupported Accept header.
413-if-content-length	**I** 413 Content Too Large for unsupported Content-Length header.
415-if-content-type	**I** 415 Unsupported Media Type for unsupported Content-Type header.
422-if-params	**I** 422 Unprocessable Content if there are parameters (semantics).
422-if-payload	**I** 422 Unprocessable Content if there is a payload (semantics).
no-200-if-error	**NI** 200 OK for a response that describes an error.
no-201-if-delete	**NI** 201 Created in a DELETE method.
no-201-if-get	**NI** 201 Created in a GET method.
no-201-if-patch	**NI** 201 Created in a PATCH method.
no-204-if-content	**NI** 204 No Content for a response that has content.
no-401-if-no-auth	**NI** 401 Unauthorized if there is no authentication mechanism.
no-403-if-no-401	**NI** 403 Forbidden if there is no 401 Unauthorized.
no-413-if-no-payload	**NI** 413 Content Too Large if there is no payload.
no-415-if-no-payload	**NI** 415 Unsupported Media Type if there is no payload.
no-non-standard-codes	**NI** non-standard status codes.

4.3 RQ.3 - Status Codes Misuses in REST APIs

After defining the status code usage rules, we implemented them in SCOAS. Then, we executed the tool for all API specifications of the benchmark, checking and reporting rule violations (misuses). SCOAS found a total of 17,767 status code misuses occurring in all specifications, highlighting their omnipresence. Figure 3 illustrates the number of specifications with at least one rule violation, per rule identifier. This representation prevents bias from larger APIs, where a certain violation could have occurred multiple times in its specification only. The total number of violations can be found in our replication package. Our results indicate that incorrect and/or inconsistent use of status codes is both widespread and systematic across REST API specifications, affecting APIs of varying popularity and application domains.

We observe that some of the most frequent rule violations are related to missing client errors in the event of invalid syntax (`has-400-xx`) and/or semantics (`has-422-xx`) of request parameters/payloads. For instance, violations of the rule `has-422-if-params` occurred in 53 specifications, which is the most frequent. These results highlight a lack of implemented client errors for such cases, rendering request debugging practices difficult for clients if their request parameters/payloads are invalid.

We also see that frequent rule violations are related to missing client errors in the event of invalid request headers (violations of `has-406-xx`, `has-413-xx`, and `has-415-xx` occurred in 36 to 39 specifications). These results suggest that header-based responses are less documented in REST APIs, perhaps as they are less frequent/known compared to parameters and payloads. Nonetheless, documenting client errors for headers remains important for related debugging.

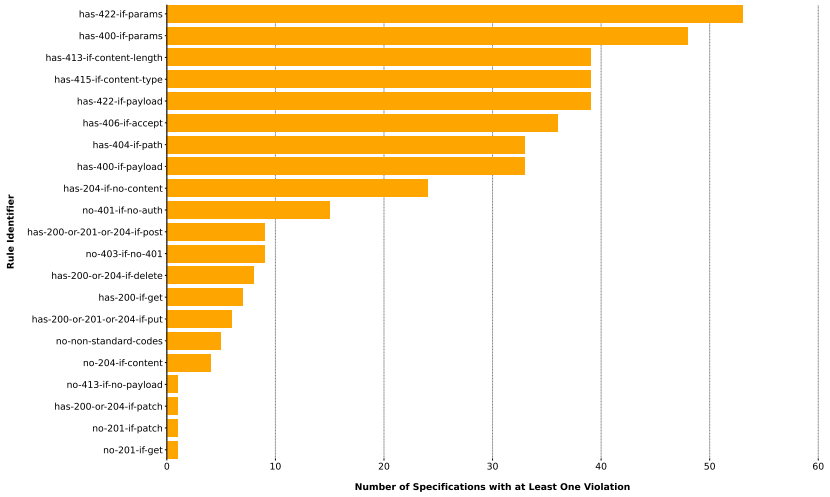

Fig. 3. Number of specifications with at least one violation per rule identifier.

In 24 specifications, violations of the rule `has-204-if-no-content` were reported, as empty and successful responses did not use the `204 No Content` status code (intended for such purpose). This result suggests that developers either forget to document response content, or forget to use the `204 No Content` status code for successful and empty responses. Conversely, four specifications used `204 No Content` in non-empty responses (violations of `no-204-if-content`), revealing more problematic misuses of the status code.

In 15 specifications, `401 Unauthorized` responses were implemented without documented authentication (violations of `no-401-if-no-auth`). Moreover, `403 Forbidden` responses were implemented without `401 Unauthorized` in nine specifications (violations of `no-403-if-no-401`). These results suggest that security/authentication mechanisms are often incorrectly documented or incomplete in REST API specifications. Yet, the OAS standard supports authentication descriptions through the `security`/`securitySchemes` fields.

More severe violations were found, such as the use of non-standard status codes (in five specifications), missing `200 OK` responses in `GET` methods (in seven specifications), and the use of `201 Created` in HTTP methods which cannot create data (in two specifications). These results suggest a lack of knowledge/API-specific behaviors deviating from HTTP standards.

We also report the average number of violations per route (vpr) for all APIs. Figure 4 illustrates this result, with bar colors indicating API size measured by the number of routes. The size is categorized into five percentile-based bins: the bottom 10% (p10), the top 10% (p90), and the middle 80% divided evenly into three bins. These bins correspond to the following categories: micro, small, medium, large, and very large APIs. As shown, very large APIs always have over 3.5vpr, while micro APIs never exceed the 3vpr threshold. This result suggests

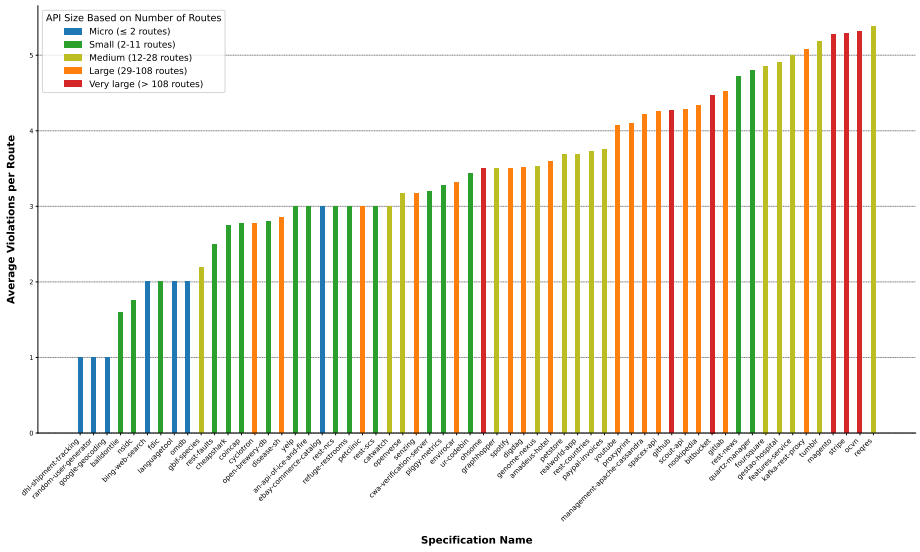

Fig. 4. Average number of violations per route for all REST API specifications of the benchmark. Bars colors represent API size in terms of routes.

that status codes are implemented more accurately in smaller APIs, potentially due to a more manageable implementation complexity. However, some small and medium APIs display a higher vpr ratio compared to some large and very large APIs, suggesting that API size alone does not fully determine the prevalence of status code misuses.

> **RQ.3 Summary**: SCOAS shown that status code misuses are frequent, with 17,767 rule violations detected across 60 specifications. Frequent issues include missing client error responses and using security-related status codes without authentication definitions. Violations per route tend to increase with API size, but exceptions in small and medium APIs indicate that size alone does not determine misuse prevalence. Violations occurred in widely used APIs (e.g., GitHub, PayPal, Spotify, Stripe), indicating that misuses are not limited to smaller or less mature projects.

5 Discussion

Implications and Recommendations for Testing Tools. Testing tools for REST APIs usually rely on status code ranges to determine the validity of test cases and to find server errors: `2xx` for valid requests, `4xx` for client errors, and `5xx` for server errors. This reliance on status codes is widespread in state-of-the-art tools; A survey by Golmohammadi, Zhang, and Arcuri [13] highlights that

`5xx` status codes are used to identify faults in over 30 different works on REST API testing. However, misusing status codes can lead to false positives and false negatives when reporting testing tools results. For instance, if a server responds with `5xx` status codes for client errors, this leads to false positives where client errors are invalidly flagged as server errors by the testing tool. Such a case misleads the tester, as this client error "disguised" as a server error is unlikely to exhibit an interesting behavior. Moreover, if a server uses `2xx` or `4xx` status codes to represent server errors, this leads to false negatives where server errors are invalidly flagged as either valid requests or client errors. The same behavior can happen with client errors being flagged as valid requests if the API always responds with `2xx` status codes. This behavior is not solely theoretical, as it can be observed in popular APIs. For instance, when sending a request with an invalid path to the Deezer API, a `404 Not Found` client error is expected. However, in this particular API, the server replies with `200 OK`. The status code seems to indicate a success, however the error is actually described in the body with an API-defined `600` code. This case is confusing, as it is in the `2xx` (successful) range, resulting in a false negative for a client error. Additionally, if no error description is provided, external users would fail to understand the meaning of the `600` code. Similarly, when implementing REST APIs with popular frameworks such as Spring, false negatives for server errors can be observed. For instance, the popular Spring Petclinic REST API returns `500 Internal Server Error` status codes when a path does not exist (i.e., `NoResourceFoundException`). While it is possible to specify status code mappings in Spring, they can be omitted and thus the `500` code is used as a fallback.

Table 3. State-of-the-art testing tools for REST APIs and their respective oracles.

Tool Name	Reference	Oracle
bBOXRT	[19]	Status codes, responses, test metadata
EvoMaster	[1]	Status codes, AI
QuickREST	[17]	Status codes, specification conformance
Morest	[20]	Status codes
Restats	[6]	Status codes
RestCT	[28]	Status codes
RESTler	[2]	Status codes, predefined checkers
RESTest	[21]	Status codes, response validation
RestTestGen	[27]	Status codes, schema match
Schemathesis	[14]	Status codes, semantic checks

To illustrate the prevalence of status code reliance in API testing tools, we analyzed the oracles of state-of-the-art testing tools for REST APIs. We selected open-source tools appearing in a survey on REST API testing [13]. Table 3 presents the testing tools and their respective oracles. As shown, some tools incorporate additional checks in their oracles, such as specification conformance, response validation, checkers, etc. However, all tools do rely on status codes (i.e., `5xx` server error codes) to find bugs. While this is perfectly acceptable as it follows HTTP standards, it highlights one possible detrimental effect of misusing status codes.

Recommendation for API Testers: To prevent invalid results due to status code misuses, testing tools should always analyze responses in depth and not only rely on status codes. For instance, response messages should be parsed to check if there are error descriptions. Moreover, such tools could setup manual status code mappings when APIs under test implement their own status codes.

Implications and Recommendations for Clients. Another implication of misusing status codes is that API clients are prone to receiving ambiguous responses from servers. For instance, in a microservice architecture with loosely coupled REST APIs, the services would rely on HTTP standards to understand responses. However, if you integrate a service that always responds with `200 OK` status codes, other services following HTTP standards would interpret all requests as valid. Doing so could cause various problems, such as rendering errors on pages, propagating invalid responses to downstream services, or logging misleading success messages. Status code misuses also introduce other client-related problems. Without precise client errors codes from the server, debugging invalid request becomes a difficult task. For instance, a server could respond with overly generic `400 Bad Request` status codes, instead of more specific codes such as `401 Unauthorized` (for a lack of authentication) or `413 Content Too Large` (for unsupported content size). Moreover, front-end applications often use status codes to display messages (e.g., "Saved successfully" for `201 Created`). If the wrong code is used, users might see misleading feedback. In consequence, handling status code misuses is also relevant for REST API clients (e.g., users and dependent microservices).

Recommendation for API Clients: REST API clients should not directly trust status codes in responses. Instead, clients should first verify the documentation (if available) of leveraged REST APIs for potential API-dependent codes or API-defined behaviors. Similarly to testers, clients should also analyze response messages.

OpenAPI Specification as Baseline. In this paper, we hypothesized that *REST API specifications perfectly reflect their implementations*, notably to assess the implications in practice. We acknowledge that APIs may return codes that are different to what is presented in the specification. Yet, the goal of this paper was not to analyze conformance issues between a specification and its implementation (left for future work). In contrast, we solely analyzed OAS files as it is very fast and does not require access to the underlying source code, which is not always public (e.g., Spotify). Moreover, official OAS guidelines suggest using a *design-first* approach [26], supporting the use of OAS as baseline. Additionally, existing and state-of-the-art tools can generate API source code from OAS files (e.g., OpenAPI Generator [22]). In consequence, using the OAS format is a good fit to standardize SCOAS for status code misuse detection in REST APIs.

6 Threats to Validity

Internal Validity. (1) The ground truth for status code usage rules was derived through manual analysis, which may be prone to human errors or subjective disagreements about what constitutes a status code misuse. To mitigate this threat, we analyzed HTTP standards (status codes and methods) defined in the official IANA registry, documented their sources from official RFCs, and collected the distribution of status codes across individual REST APIs (cf. Section 4.1 and Section 4.2). This process assisted us in generating relevant rules based on typical REST interactions and CRUD operations. However, we acknowledge that the rules may not capture every relevant consideration for status codes in REST APIs, as their applications may depend on specific contextual factors. **(2)** The implementation of our tool may be prone to issues or bugs. We mitigated this threat with validation, testing, and by cross-checking outputs.

External Validity. (1) The benchmark of REST API specifications may not fully reflect the diversity of real-world API specifications, which introduces potential construction bias. To mitigate this threat, we utilized a peer-reviewed benchmark [8] containing 60 REST APIs varying in popularity, size, and application domain. Moreover, specifications may have been updated since our evaluation, creating inconsistencies between the versions analyzed and their current state. **(2)** Our study was limited to using the OpenAPI Specification format to document REST APIs. Although this format is the most widely used, we acknowledge this as a limitation regarding other existing documentation formats for REST APIs (e.g., Postman Collections, RAML, API Blueprint). **(3)** To mitigate threats related to the practical usefulness of our tool, we designed SCOAS to be lightweight, being able to run on a "standard" laptop in a couple of seconds (cf. Section 4). We also opted for input/outputs in JSON, HTML, and LOG file formats, which are widely used in practice.

7 Conclusion and Future Work

We presented SCOAS (Status Code Analysis in OpenAPI Specifications), a novel tool aimed at detecting status code misuses in REST API specifications. Our tool takes an OpenAPI Specification (OAS) file as input and generates status code contexts based on paths, methods, and responses described in the specification. By comparing contexts against a set of defined status code usage rules, violations can be detected and reported. Our evaluation demonstrated that status code misuses are both frequent and systematic in REST APIs, with a total of 17,767 rule violations detected across 60 REST API specifications. We highlighted practical implications of status code misuses related to testing, clients, and microservices. We also provided recommendations for API testers and clients.

For future work, we plan to keep our tool up-to-date and add new status code usage rules by analyzing the evolution of HTTP standards and REST APIs. We would also like to analyze and fix status code misuses directly in REST API implementations.

Use of Generative AI. Generative AI was used for limited rephrasing purposes. The authors reviewed the content and take full responsibility for it.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Arcuri, A.: RESTful API Automated Test Case Generation with EvoMaster. ACM Trans. Softw. Eng. Methodol. **28**(1) (Jan 2019). https://doi.org/10.1145/3293455
2. Atlidakis, V., Godefroid, P., Polishchuk, M.: Restler: Stateful rest api fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 748–758. IEEE (2019)
3. Benhabbour, I., Attia, M., Dacier, M.: HTTP Conformance vs. Middleboxes: Identifying Where the Rules Actually Break Down. In: International Conference on Passive and Active Network Measurement. pp. 155–181. Springer (2025)
4. Bogner, J., Kotstein, S., Abajirov, D., Ernst, T., Merkel, M.: RESTRuler: Towards Automatically Identifying Violations of RESTful Design Rules in Web APIs. In: 2024 IEEE 21st International Conference on Software Architecture (ICSA). pp. 123–134 (2024). https://doi.org/10.1109/ICSA59870.2024.00020
5. Corradini, D., Montolli, Z., Pasqua, M., Ceccato, M.: 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. p. 1383–1394. ASE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3691620.3695511
6. Corradini, D., Zampieri, A., Pasqua, M., Ceccato, M.: Restats: A test coverage tool for RESTful APIs. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 594–598. IEEE (2021)
7. Decrop, A.: SCOAS (2026), https://github.com/alixdecr/scoas
8. Decrop, A., Eraso, S., Devroey, X., Perrouin, G.: A Public Benchmark of REST APIs. In: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR). pp. 421–433. IEEE Computer Society, Los Alamitos, CA, USA (Apr 2025). https://doi.org/10.1109/MSR66628.2025.00072
9. Di Meglio, S., Pontillo, V., Starace, L.L.L.: REST in Pieces: RESTful Design Rule Violations in Student-Built Web Apps. arXiv preprint arXiv:2507.11689 (2025)
10. Ed-douibi, H., Cánovas Izquierdo, J.L., Cabot, J.: Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In: 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC). pp. 181–190 (2018). https://doi.org/10.1109/EDOC.2018.00031
11. Fielding, R., Nottingham, M., Reschke, J.: RFC 9110: HTTP Semantics (2022), https://www.rfc-editor.org/rfc/rfc9110.html
12. Fielding, R.T.: Architectural styles and the design of network-based software architectures. University of California, Irvine (2000)
13. Golmohammadi, A., Zhang, M., Arcuri, A.: Testing RESTful APIs: A Survey. ACM Trans. Softw. Eng. Methodol. **33**(1) (Nov 2023). https://doi.org/10.1145/3617175

14. Hatfield-Dodds, Z., Dygalo, D.: Deriving semantics-aware fuzzers from web API schemas. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. p. 345–346. ICSE '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510454.3528637
15. Huang, R., Motwani, M., Martinez, I., Orso, A.: Generating REST API Specifications through Static Analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639137
16. Internet Assigned Numbers Authority (IANA): Hypertext Transfer Protocol (HTTP) Status Code Registry (2025), https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml
17. Karlsson, S., Čaušević, A., Sundmark, D.: QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). pp. 131–141. IEEE (2020)
18. Kim, M., Sinha, S., Orso, A.: LlamaRestTest: Effective REST API Testing with Small Language Models. Proc. ACM Softw. Eng. **2**(FSE) (Jun 2025). https://doi.org/10.1145/3715737
19. Laranjeiro, N., Agnelo, J., Bernardino, J.: A Black Box Tool for Robustness Testing of REST Services. IEEE Access **9**, 24738–24754 (2021). https://doi.org/10.1109/ACCESS.2021.3056505
20. Liu, Y., Li, Y., Deng, G., Liu, Y., Wan, R., Wu, R., Ji, D., Xu, S., Bao, M.: Morest: model-based RESTful API testing with execution feedback. In: Proceedings of the 44th International Conference on Software Engineering. p. 1406–1417. ICSE '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510133
21. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: RESTest: automated black-box testing of RESTful web APIs. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 682–685 (2021)
22. OpenAPI Generator Team: OpenAPI Generator (2026), https://github.com/OpenAPITools/openapi-generator
23. Rautenstrauch, J., Stock, B.: Who's Breaking the Rules? Studying Conformance to the HTTP Specifications and its Security Impact. In: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security. pp. 843–855 (2024)
24. Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L., Percannella, G.: REST APIs: A large-scale analysis of compliance with principles and best practices. In: International conference on web engineering. pp. 21–39. Springer (2016)
25. SmartBear Software: Swagger Editor (2026), https://editor.swagger.io
26. The Linux Foundation: OpenAPI Initiative (2026), https://www.openapis.org
27. Viglianisi, E., Dallago, M., Ceccato, M.: RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). pp. 142–152 (2020). https://doi.org/10.1109/ICST46399.2020.00024
28. Wu, H., Xu, L., Niu, X., Nie, C.: Combinatorial testing of RESTful APIs. In: Proceedings of the 44th International Conference on Software Engineering. p. 426–437. ICSE '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510151