Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

UNIVERSITÉ DE NAMUR

researchportal.unamur.be

University of Namur

RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

Analyzing Status Code Misuses in REST API Specifications

Decrop, Alix; Papadakis, Mike; Perrouin, Gilles

Published in:
Proceedings of the 26th International Conference on Web Engineering, ICWE 2026

Publication date:
2026

Document Version
Peer reviewed version

Link to publication

Citation for pulished version (HARVARD):
Decrop, A, Papadakis, M & Perrouin, G 2026, Analyzing Status Code Misuses in REST API Specifications. in *Proceedings of the 26th International Conference on Web Engineering, ICWE 2026.*

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. May. 2026

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

Final Accepted

Abstract. Many web APIs rely on the REST architectural style, exploiting HTTP for client-server interactions. Clients typically interpret server responses using status codes, such as `200 OK` (success) or `404 Not Found` (unavailable resource). As REST does not enforce HTTP standards, servers may misuse status codes (e.g., using `500` for a client error instead of `4xx`). Such misuses cause false positives during testing and confuse clients. In this paper, we explore the prevalence of status code misuses in REST APIs. We introduce SCOAS, a tool that detects status code misuses in OpenAPI specifications using 24 rules derived from HTTP standards. We demonstrate that status code misuses are systematic in REST APIs, with 17,767 rule violations detected across 60 specifications. We further discuss their implications and provide recommendations for testers and clients.

Keywords: REST API · OpenAPI · Status Code · Misuse

1 Introduction

Web APIs widely rely on the REpresentational State Transfer (REST) architectural style [8], which uses HTTP for stateless client-server communications. In this context, HTTP status codes are used to indicate the outcome of requests. Typical status codes indicate successes (`2xx`), client errors (`4xx`), and server errors (`5xx`). The OpenAPI Specification (OAS) [13] standard is widely used to document these behaviors, allowing clients and testers to understand API responses correctly. However, this paper finds that REST APIs deviate from status code semantics. We term such deviations *status codes misuses*, which are not benign. For instance, using `2xx` or `5xx` status codes to represent client errors misleads clients, undermines interoperability, and reduces the accuracy of testing tools. Moreover, the use of non-standard or overly generic status codes (e.g., `400` for advanced client errors) hinders debugging due to imprecise/misleading feedback.

REST API design conformance is an active field; Di Meglio et al. [6] analyzed RESTful design rule violations in web apps. Bogner et al. [2] implemented RESTRuler, a tool aimed at detecting design rule violations in OpenAPI descriptions,

based on Massé's book on REST API design [10]. Rodriguez et al. [12] analyzed the compliance of HTTP traffic with REST API principles and best practices. Existing tools [1,3,14] can analyze OpenAPI specification-implementation conformance by checking mismatches and/or undeclared data. Yet, such approaches require source code, do not cover fine-grained misuses, and/or do not focus specifically on HTTP status code semantics.

Therefore, our work investigates HTTP status code misuses in REST APIs at the specification-level, whose prevalence has not been explored so far. We offer the following novel contributions: **(1)** SCOAS, a tool for Status Code Analysis in OpenAPI Specifications. **(2)** A set of 24 status code usage rules, derived from HTTP standards and REST API design/usage. **(3)** A study of status code misuses in 60 REST API specifications. A total of 17,767 misuses were identified, occurring in various APIs (GitHub, Spotify, Stripe, etc.). **(4)** A replication package containing our implementation and evaluation data [4].

2 Approach

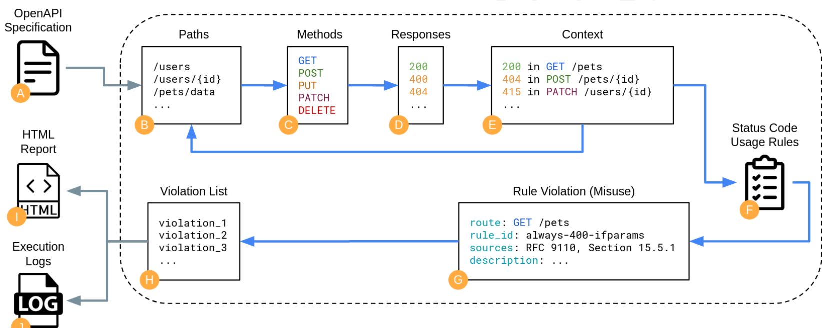

Fig. 1. Overview of our tool SCOAS.

To detect status code misuses, we developed SCOAS (Status Code Analysis in OpenAPI Specifications). Figure 1 illustrates our tool, with orange circles X representing each phase. First, an OAS file (JSON format) of a REST API is provided A. Then, we iterate over all paths specified in the file 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 required, as status code usage rules may 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 3). We verify if rules are

violated by performing a static analysis of the context against the rules. If it is the case, we generate a rule violation (misuse) containing the triggering context and related information G. We append it to a list of violations H. The process continues until all status code contexts have been analyzed. SCOAS stops when the iterations are completed (B - E). Upon ending, SCOAS issues an HTML report I and execution logs J.

3 Evaluation

We formulate the following research question: *What is the prevalence of status code misuses in REST API specifications?* By answering it, we aim to analyze and discuss the prevalence of status code misuses in real-world REST API specifications using SCOAS.

Setup. We formed a comprehensive dataset of 60 unique specifications (in the OAS format) from various REST APIs. These specifications were extracted from the Public REST API Benchmark (PRAB) [5], containing diverse APIs (GitHub, Language Tool, Petclinic, etc.). The complete list of APIs along with their structural data can be found in the PRAB repository. Our evaluation was conducted using a laptop with a 2.4GHz processor and 16GB of RAM. SCOAS is deterministic, operates offline, and processes a specification within seconds.

Status Code Distribution. First, we identified which HTTP status codes are commonly used in REST API responses, and defined a set of relevant usage rules. We analyzed the specifications included in our benchmark dataset, and extracted all occurrences of status codes. Figure 2 illustrates the distribution of status codes per REST API. In total, we identified 35 distinct status codes, accounting for 12,028 occurrences. `2xx` and `4xx` are the most frequent ranges (`200 OK` and `404 Not Found` being the most frequent status codes), followed by `5xx` which occurs marginally. We observed a lack of `1xx` and `3xx` codes, less suitable for typical REST interactions [8]. We also identified non-standard status codes (e.g., `0`, `420`, `555`), suggesting API-dependent behaviors.

Status Code Usage Rules. Based on our findings, we defined 24 unique status code usage rules for REST APIs. We selected occurring status codes from the `2xx` range (`200`, `201`, `204`) and the `4xx` range (`400`, `401`, `403`, `404`, `406`, `413`, `415`, `422`), as they are sufficient for typical client-server interactions [4,8,10]. We excluded the `5xx` range, as it describes errors unrelated to an "expected" interaction (e.g., server maintenance or bad gateway). We defined our rules based on a structured manual analysis of official HTTP standards [7], REST API principles [8], and best practices [10]. Notably, we identified statement verbs (e.g., "shall", "must not", etc.) and mapped them to adequate rules for REST API responses. Table 1 presents a subset of the rules (the full list can be found in our replication package [4]).

Status Code Misuses. We implemented the rules and executed our tool to check for rule violations (misuses) in REST API specifications. SCOAS analyzed a total of 12,028 status codes across 4,016 routes, and found 17,767 status code misuses (occurring in all specifications, highlighting their omnipresence). Figure 3

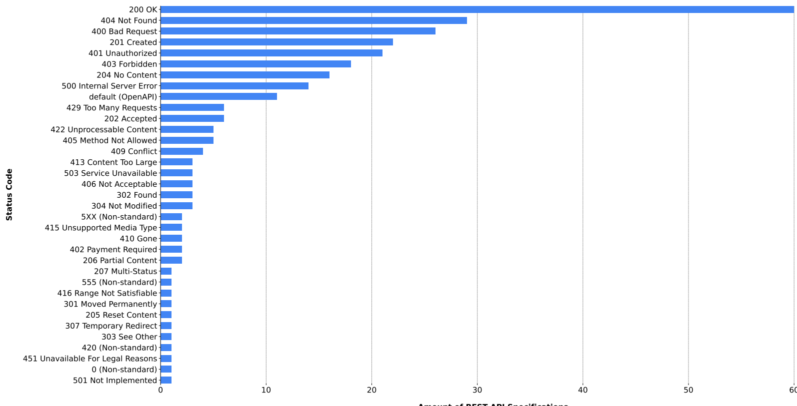

Fig. 2. Distribution of HTTP status codes across REST API specifications.

Table 1. Subset of status code usage rules. **I** = Implement, **NI** = Never Implement.

Identifier	Description
`200-if-get`	**I** `200 OK` in a `GET` 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).
`404-if-path`	**I** `404 Not Found` if there are path parameters.
`406-if-accept`	**I** `406 Not Acceptable` for unsupported `Accept` header.
`422-if-payload`	**I** `422 Unprocessable Content` if there is a payload (semantics).
`no-201-if-get`	**NI** `201 Created` in a `GET` 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-413-if-no-payload`	**NI** `413 Content Too Large` if there is no payload.
`no-non-standard-codes`	**NI** non-standard status codes.

illustrates the number of specifications with at least one rule violation, per rule identifier. Our results indicate that the incorrect and inconsistent use of status codes is systematic in REST API specifications.

We observe that frequent rule violations are related to missing client errors for invalid request syntax/semantics. Indeed, violations of the rule `422-if-params` occurred in 53 specifications; A lack of implemented client errors complicates request debugging. Moreover, violations of rules related to client errors for invalid request headers occurred in up to 39 specifications. This suggests that header-based responses are less documented in REST APIs, perhaps as they are less used compared to parameters and payloads. Nonetheless, documenting such errors remains important as headers are used in REST APIs [11]. In 24 specifications, violations of the rule `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 indicates that developers either forget to document response content, or forget to use the `204 No Content` status code for successful and empty responses. Conversely, 4 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

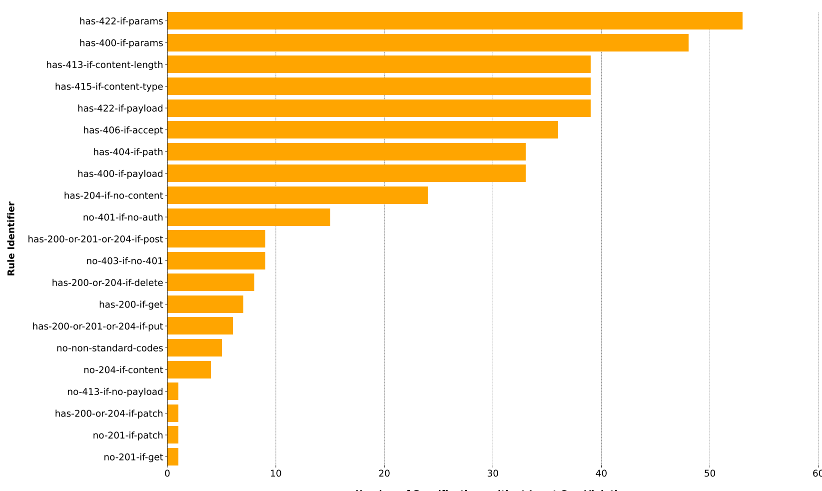

Fig. 3. Distribution of rule violation occurrence across REST API specifications.

documented authentication (violations of `no-401-if-no-auth`). Moreover, `403 Forbidden` responses were implemented without `401 Unauthorized` in 9 specifications (violations of `no-403-if-no-401`), suggesting that security mechanisms are often incorrectly documented in REST API specifications. Yet, the OAS standard supports security descriptions through the `security`/`securitySchemes` fields [13]. More severe violations were found, such as the use of non-standard status codes (in 5 specifications), missing `200 OK` responses in `GET` methods (in 7 specifications), and the use of `201 Created` in HTTP methods which cannot create data (in 2 specifications). In consequence, these violations suggest insufficient knowledge of HTTP standards and non-conformance to REST API design rules [10].

We also report the mean 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 5 percentile-based bins: the bottom 10% (p10), the top 10% (p90), and the middle 80% divided evenly into 3 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 that status codes are used more accurately in smaller APIs, potentially due to a lower 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.

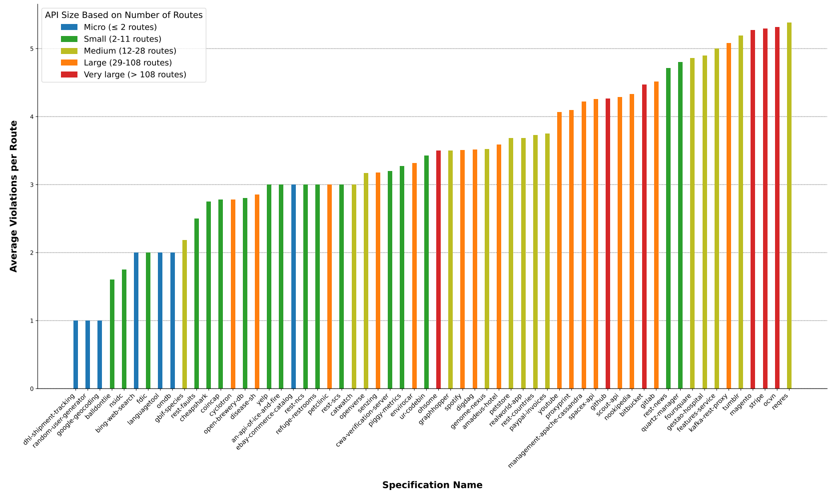

Fig. 4. Mean violations per route for REST API specifications, colored by size.

Summary: Status code misuses are frequent in REST APIs, with 17,767 rule violations detected across 60 specifications. Common issues include missing client error responses and using security-related status codes without security definitions. Violations per route tend to increase with API size, yet exceptions in smaller 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.

4 Discussion

Implications for Testing Tools. REST API testing tools rely on status code ranges for test outcomes and to find server errors: `2xx` for valid requests, `4xx` for client errors, and `5xx` for server errors. Golmohammadi et al. [9] highlight that `5xx` status codes are used to identify faults in over 30 different works on REST API testing. Thus, misusing status codes can lead to false positives/negatives in such tools. For instance, if a server responds with `5xx` status codes for client errors, this leads to false positives (inaccurate bug report). This misleads the tester, as this false server error is unlikely to exhibit an interesting behavior. Similarly, if an API uses `2xx` or `4xx` status codes to represent server errors, this leads to false negatives. This can also happen with client errors if the API always responds with `2xx` status codes. For instance, when sending a request with an invalid path to the Deezer API, a `404 Not Found` client error is expected. However, the API replies with `200 OK`. While this status code indicates a success, an error message is found in the response body with an API-defined `600` code. This deviates from REST API design (non-standard status codes), and confuses

tools if no additional response parsing is performed. Similarly, false positives can be observed in popular API frameworks such as Spring. For instance, the Spring Petclinic REST API returns `500 Internal Server Error` status codes when a path does not exist (i.e., `NoResourceFoundException`), instead of `404 Not Found`. 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.

> **Recommendations for Testers**: REST API testing tools should always analyze responses in depth to avoid false positives/negatives from status codes. Response messages should be parsed to check for errors, and status code mappings should be set up when APIs implement their own codes.

Implications 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 a service 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 requests 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 see misleading feedback.

> **Recommendations for Clients**: REST API clients should not directly trust status codes. Instead, clients should check documentation for potential API-dependent codes or API-defined behaviors. Similarly to testers, clients should also analyze response content and messages in-depth.

OAS as Baseline. We hypothesized that *REST API specifications perfectly reflect their implementations*, notably to assess the implications in practice. We acknowledge that REST APIs may behave differently in practice, which is not considered in the scope of this paper and is left for future work.

5 Conclusion and Future Work

In this work, we explored the prevalence of status code misuses in REST APIs. To do so, we developed SCOAS, a tool aimed at detecting status code misuses in OpenAPI specifications. The tool operates by comparing API responses against 24 status code usage rules derived from HTTP standards, and reports potential violations. Our evaluation showed that a total of 17,767 rule violations were

detected across 60 specifications, highlighting the omnipresence of status code misuses in REST APIs. This led us to provide insights and relevant implications/recommendations for API testers and clients alike. For future work, we plan to add new status code usage rules (e.g., for the `3xx` range) by analyzing the evolution of HTTP standards and REST APIs. We also plan to add automated rule fixes for OAS files in SCOAS, and expand the work for dynamic analysis.

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.

Acknowledgments. Gilles Perrouin is an FNRS Research Associate.

References

1. Aptori: OpenAPI conformance analyzer (2025), `https://docs.aptori.dev/sift/analyzers/openapi-conformance`
2. 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) (2024)
3. Bradburn, D.: OpenAPI conformance (2019), `https://github.com/crunchr/openapi_conformance`
4. Decrop, A.: SCOAS (2026), `https://github.com/alixdecr/scoas`
5. 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)
6. 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)
7. Fielding, R., Nottingham, M., Reschke, J.: RFC 9110: HTTP semantics (2022), `https://www.rfc-editor.org/rfc/rfc9110.html`
8. Fielding, R.T.: Architectural styles and the design of network-based software architectures. University of California, Irvine (2000)
9. Golmohammadi, A., Zhang, M., Arcuri, A.: Testing RESTful APIs: A survey. ACM Trans. Softw. Eng. Methodol. **33**(1) (Nov 2023)
10. Massé, M.: REST API design rulebook. O'Reilly Media, Inc. (2012)
11. Neumann, A., Laranjeiro, N., Bernardino, J.: An analysis of public REST web service APIs. IEEE Transactions on Services Computing **14**(4), 957–970 (2018)
12. 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. Springer (2016)
13. SmartBear: OpenAPI specification (2026), `https://swagger.io/specification`
14. Specmantic: Specmantic (2026), `https://github.com/specmatic/specmatic`

Just Accepted