Defects4REST: A Benchmark of Real-World Defects to Enable Controlled Testing and Debugging Studies for REST APIs

Rahil P. Mehta
Oregon State University
Corvallis, Oregon, USA
mehtara@oregonstate.edu

Pushpak Katkhede
Oregon State University
Corvallis, Oregon, USA
katkhedp@oregonstate.edu

Manish Motwani
Oregon State University
Corvallis, Oregon, USA
motwanim@oregonstate.edu

ABSTRACT

Existing research on REST API testing and debugging often relies on arbitrarily selected APIs for evaluation, leaving a critical gap in our understanding of real world defects in REST API based projects, particularly those that can be automatically detected or repaired. To address this gap, we present Defects4REST, the first public benchmark of defects in REST API projects, systematically mined from open source GitHub repositories. Using a semi-automated process that combines clustering, topic modeling, and manual validation, we build a dataset of 607 REST API defects from 50 projects and organize them into a taxonomy of 6 defect types and 13 sub-types. Each defect is linked to an issue report, the developer-modified versions of the patched files, and the corresponding commit messages. Defects4REST features easy access to the buggy and patched versions of projects for each defect and enables analysis of the associated artifacts. To evaluate current automated testing capabilities, we assess four state-of-the-art REST API testing tools on a sample of 30 defects from 11 projects. Our evaluation reveals that tools detect only 10% (3/30) of real-world defects due to nine fundamental limitations including semantic validation, stateful testing, and infrastructure-aware testing. We release both the benchmark and the issue mining framework to support reproducible research in REST APIs, enabling the development and evaluation of REST API testing and debugging techniques.

KEYWORDS

REST API, defect benchmark, API testing, issue classification, LLM

ACM Reference Format:
Rahil P. Mehta, Pushpak Katkhede, and Manish Motwani. 2026. Defects4REST: A Benchmark of Real-World Defects to Enable Controlled Testing and Debugging Studies for REST APIs. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3787787

1 INTRODUCTION

Web Application Programming Interfaces (APIs) allow services to be accessed over the network. RESTful (or REST) APIs use the REpresentation State Transfer (REST) [4] protocol for communicating between distributed services, and are a popular type of web API used in modern software systems. Given their critical role, ensuring their reliability and correctness is essential. Accordingly, extensive research is done in developing automated testing techniques to detect issues in REST API-based projects [8]. However, these techniques primarily focus on detecting functional errors resulting in $5XX$ response codes without focusing on the specific defect types that cause those errors. Thus, **it is unknown to what extent the existing automated testing and debugging tools capture the breadth of real-world REST API failures encountered in practice**. A precursor to answer this is having a standardized defect benchmark of real-world failures in REST APIs that are beyond those detected by automated tools. Unfortunately, there does not exist such defect benchmark for REST API-based applications.

To address this problem, we introduce Defects4REST, the first public benchmark of real-world REST API defects, systematically mined from open-source GitHub repositories of REST-API-based applications. Furthermore, we use our benchmark to evaluate and compare four state-of-the-art REST API testing tools (Schemathesis [10], RESTler [2], EvoMaster [1], and AutoRestTest [25]), providing new insights into their strengths, weaknesses, and opportunities for improvement.

To construct the benchmark, we first selected a comprehensive and diverse set of 51 real-world REST API-based project repositories from GitHub using specific criteria along with including the APIs used in prior REST API research. For the selected APIs, we then mined and created a dataset of 11,313 closed issue reports that are associated with modified files and commit messages. Next, we used LLM-based classifier and manual inspection to filter out the reports that are not related to REST APIs and constructed a dataset of 607 REST API defects. Finally, we used a semi-automated approach that uses clustering and topic modeling followed by manual validation to derive a novel taxonomy of 6 defect types and 13 sub types, covering issues such as schema validation and query parameter handling errors, misconfigurations and environment-specific bugs, authentication and token management issues, middleware and runtime failures, data storage and access problems, and rare distributed system coordination faults. Since our taxonomy is derived from real-world defects and is not biased by the faults detected by existing techniques, it captures a broader and more diverse set of faults encountered in real-world REST API development.

In addition to categorizing the 607 defects using our taxonomy, we release Defects4REST v1.0 [14], a framework inspired by the Defects4J [12]. This framework enables checking out and setting up buggy or patched versions of 12 REST API projects corresponding to a subset of 110 defects that we could manually replicate, and adding more APIs and defects. Users can analyze associated issue artifacts and execute testing techniques on these real-world defects. Defects4REST provides a solid empirical foundation for REST API

ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787787

testing and debugging research, allowing evaluation of automated approaches against authentic failures encountered in practice.

In this study, we answer the following research questions:

RQ1: What types of defects commonly occur in real-world REST APIs? Analyzing the types of 607 defects from 50 projects revealed that the most defects (42%) involve data validation and query processing, followed by configuration and environment issues (27%). Authentication, data storage, and runtime errors each account for 10%, while distributed system failures are rare (1%). Overall, input validation and configuration are the most common sources of REST API defects.

RQ2: For each REST API defect type, what kinds of files are most frequently modified during fixes, and how soon do developers resolve the issues? Analyzing the frequency of eight types of patched files associated with issues, we found that source files are the most frequently modified across all REST API defect types, often alongside configuration, test, and documentation files. Defects in REST API Routing and Response Handling tend to take longer to fix, while Volume and File Handling Errors are typically resolved more quickly.

RQ3: How effectively do state-of-the-art automated REST API testing tools detect real-world defects, and what types of faults remain beyond their reach? Our evaluation of four state-of-the-art REST API testing tools (Schemathesis, RESTler, EvoMaster, and AutoRestTest) on a representative sample of 30 real-world defects from 11 APIs shows that these tools detect only 10% (3/30) of the defects and could potentially detect an additional 30% (9/30) with longer execution time, improved input generation, or more complete OpenAPI specifications. However, they fundamentally cannot detect the remaining 60% (18/30) of defects because they lack support for authentication handling, infrastructure dependencies, session management, semantic correctness, and business logic validation.

Overall, this paper makes the following key contributions:

- A novel methodology and framework to mine diverse real-world REST API defects from open-source projects on GitHub and generate a taxonomy from the mined defects.
- A manually curated dataset of 607 REST API defects obtained from 50 open-source real-world projects, categorized into 6 defect types and 13 sub types.
- Defects4REST v1.0 [14], the first public benchmark of 110 defects from 12 real-world open-source REST API projects, supporting controlled testing and debugging of REST APIs.
- An evaluation of four state-of-the-art REST API testing tools on a representative subset of Defects4REST, revealing consistent gaps across testing paradigms (schema-based, model-based, evolutionary, and learning-based) and offering actionable implications and concrete research directions for advancing REST API testing and debugging.
- A replication package for our paper containing the code and data is available at https://github.com/ANSWER-OSU/Defects4REST-ReplicationPackage.

2 BACKGROUND AND RELATED WORK

Representational State Transfer (REST) is an architectural style for designing networked applications, primarily used for web services. REST APIs enable communication between distributed systems by leveraging HTTP methods such as GET, POST, PUT, DELETE, and PATCH to perform CRUD (Create, Read, Update, Delete) operations on resources. A well-designed REST API follows key principles, including statelessness, uniform interface, resource-based structure, and cacheability, ensuring scalability and ease of integration. REST APIs are widely adopted in modern software development, serving as the backbone for microservices, mobile applications, cloud computing, and third-party integrations. However, despite their widespread use, REST APIs are prone to various defects related to specification conformance, security vulnerabilities, performance bottlenecks, and inconsistent behavior. These defects can lead to faulty responses, unexpected failures, and security risks, making automated testing and defect analysis crucial for maintaining API reliability and robustness. Existing REST API testing tools focus on detecting certain classes of errors, but they often fail to capture the full spectrum of real-world defects, highlighting the need for a comprehensive defect benchmark. As APIs become increasingly complex, the need for robust testing and quality assurance methodologies has grown exponentially.

While there have been efforts to create benchmarks for API structural characteristics, **there exists no open-source defect benchmark for REST APIs**. For example, the PRAB (Public REST API Benchmark) [3], provides a collection of open-source REST APIs to support automated testing research, but it does not categorize or analyze defects present in those APIs. Similarly, EvoMaster Benchmark (EMB) [1] was originally created using arbitrarily selected or artificially created APIs to evaluate EvoMaster's effectiveness in identifying functional and security-related faults, leaving other categories of defects unexplored. Finally, RESTBench [24] provides a manually annotated dataset of two real-world scenarios to assess how effectively LLMs can handle tasks like planning, API calling, and response parsing when interacting with real-world applications (e.g., movie databases, music players) via REST APIs, but it does not focus on real-world API defects repaired by developers.

A related line of work involves empirical studies of API failures, such as the taxonomy proposed in [13], which categorizes REST API faults detected by EvoMaster in seven open-source Java APIs into categories such as Schema Conflicts, Configuration Faults, and Faults in Business Logic. However, the taxonomy is inherently limited to faults that can be identified by EvoMaster and may not reflect the broader spectrum of real-world defects encountered by developers in production REST APIs.

Finally, recent work by Zhu et al. [26] surveys 132 software defect datasets across eleven distinct application domains, with machine learning, compilers and runtime engines, and mobile applications. The survey analyzes five critical dimensions: scope (application domains, defect types, programming languages), construction methodologies (data sources and extraction methods), availability and usability, practical usage patterns, and future research opportunities. Despite the comprehensive coverage of 132 software defect datasets across diverse domains and programming languages, **there exists a notable absence of dedicated defect benchmarks specifically targeting REST API defects**. This gap represents a significant oversight in the current landscape of software defect research, particularly given the ubiquity and critical importance of REST APIs in modern software architecture.

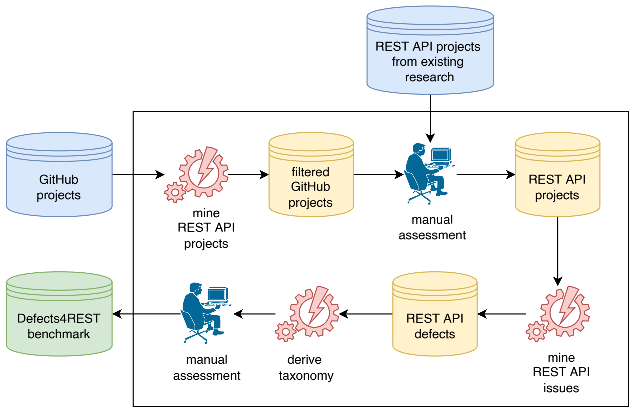

Figure 1: The Defects4REST construction methodology.

Our work addresses this gap by introducing Defects4REST, the first *executable* defect benchmark that categorizes REST API defects from a diverse set of open-source projects. Compared to prior datasets that rely primarily on keyword matching, continuous integration (CI) fail-pass patterns, or synthetic mutation, our approach introduces a novel semi-automated pipeline that leverages large language model (LLM). This benchmark provides a data-driven foundation for evaluating testing tools, improving API quality, and guiding best practices in API development and maintenance.

3 METHODOLOGY

Figure 1 shows our methodology for constructing Defects4REST benchmark. First, we mine relevant open-source REST API projects from GitHub (Section 3.1). Next, from the selected projects, we mine closed issue reports and classify them into REST API and non-REST API defects (Section 3.2). Finally, from the identified defects, we derive a defect taxonomy to characterize them based on their issue description and developer-patched files (Section 3.3).

3.1 Selecting Candidate REST API Projects

We identified real-world open-source REST API projects by using GitHub API [7] to search for repositories that have *REST API* phrase in the repository *name, description, README*, or *topics* and selected those that have ≥100 stars, ≥100 forks, and ≥5000 kb size to ensure that these repositories are widely used and are not toy projects. Next, we sorted the search results in the decreasing order of forks approximating the number of forks to represent distinct API users and considering that more users provides a higher likelihood of mining diverse issues. We manually skimmed through top-500 repositories' documentation, codebase, presence of at least one closed issue, and last update timestamps to confirm project activity and ensure the presence of meaningful REST API issue reports. Additionally, we also analyzed 60 REST API projects that are part of the recently published PRAB dataset [3] constructed by analyzing over 100 research papers published on REST APIs and included 10 projects that (1) are open-source and (2) have at least one closed issue associated with a commit. Reconciling both the analyses, we created a dataset of 51 open-source REST API projects with at least one closed issue that has a associated commit fixed by the developer. These 51 projects are implemented using 9 different programming languages (Java, JavaScript/TypeScript, Python, C#, Go, PHP, Ruby, Dart, and C++) and vary in codebase size from 5,463 to 2,604,124 lines of code.

3.2 Mining REST API Defects

The process to automatically extract REST API defects from the selected GitHub projects involved two steps: (1) mining the closed issue reports and associated metadata (Section 3.2.1) and (2) classifying the mined issues into REST API defects and non-REST API defects (Section 3.2.2).

3.2.1 Mining Closed Issues from Selected Repositories and representing them in XML format. While analyzing the closed issues, we observed that in several projects, developers created issues for their to-do tasks and were not actually issues filed by the users of the application. To filter out such issues, we mined issues that had at least one commit associated with it indicating that developers modified some code to address those issues. To mine such closed issues from the selected REST API projects, we use GitHub REST API endpoints[1]. Specifically, for each selected project, we fetched all the closed issues using the **`GET /repos/{owner}/{repo}/issues`** endpoint. Since this endpoint's response includes both closed issues and pull requests, we filtered out the pull requests by checking for the presence of the *pull_request* key and thereby only retaining entries that represent actual issues. Next, for each closed issue retained after filtering out pull requests, we retrieve the sequence of associated *events* using **`GET /repos/{owner}/{repo}/issues/{issue_number}/timeline`** endpoint. Among these events, we searched for those with type *referenced*, which indicated that a commit has referenced the issue (typically through messages like Fixes #123). If such an event included the *commit_id* field, we extracted the SHA and message associated with the commit. As developers may push more than one commits to fix an issue, we considered all the commits that referenced the same issue and involved modifying at least one file to contain the developer's repair for that issue. Using the retrieved commit SHAs, we fetched all the modified files associated with those commits. Next, to determine the buggy version of the project for an issue, we fetched the parent commit of the fixed commit that serves as the last known buggy state of the project before the developers' fix was applied. In cases where multiple commits referenced the same issue, we fetched the parent commit of the oldest commit that referenced the issue. Finally, we calculated the time to resolve the issue by computing the difference between issue's *created_at* and *closed_at* timestamps. All the extracted issues and their associated metadata including the project name, issue number, issue URL, title, description, number of days it took developers to resolve the issue, buggy commit details (SHA and message), and fixed commit details (SHA, message, and patched files) were represented and stored in XML format as shown in Figure 2 for the issue[2] extracted from the Ghost project, which is a headless content management system used for publishing content for blogs, newsletters, and online publications. Overall, the outcome of this step was a dataset of 11,313 closed issues mined from 51 projects.

[1]https://docs.github.com/en/rest
[2]https://github.com/TryGhost/Ghost/issues/14980

```xml
<?xml version="1.0" ?>
<ISSUE>
<PROJECT>Ghost</PROJECT>
<ISSUENO>14980</ISSUENO>
<ISSUEURL>
    https://github.com/TryGhost/Ghost/issues/14980
</ISSUEURL>
<TITLE>
Tenor V1 API has been decomissioned, unable to load Gifs from editor
</TITLE>
<DESCRIPTION>
### Issue Summary Whenever trying to insert a gif from Tenor on ...
### Relevant log / error output ```shell GET https://g1.tenor.com/v1
       /trending?media_filter=minimal&key=null&contentfilter=off 401
        (Unauthorized) ```...
</DESCRIPTION>
<DAYSTOFIX>47</DAYSTOFIX>
<BUGGYCOMMIT>
<MESSAGE>v5.7.1</MESSAGE>
<SHA>cf64dc89d037effb3240fcccfe34285d46a48078</SHA>
</BUGGYCOMMIT>
<PATCHCOMMITS>
<COMMIT>
<MESSAGE>
Updated Tenor API to v2 (#2418) refs: https://github.com/TryGhost/
       Ghost/issues/14980 refs: https://github.com/TryGhost/Ghost/
       pull/15087 - The Tenor v1 API has been decommissioned https
       ://developers.google.com/tenor/guides/migrate-from-v1 -
       Updated the API to v2, but there are some differences we have
        to account for - Swapped from using the old "trending" API
        to the new "featured" API, which at present seem to be the
       same thing - Added a new client_key, which identifies the
       integration using the google API key, as google API keys can
       be used for multiple APIs and projects - Fixed up the error
        ...
</MESSAGE>
<SHA>cc276486f0360eb3017e1d55f0d3dd9b2d3a2228</SHA>
<PATCHEDFILES>
    <FILE>
    ghost/admin/app/services/tenor.js
    </FILE>
    <FILE>
    ghost/admin/lib/koenig-editor/addon/components/koenig-card-image
           /selector-tenor.hbs
    </FILE>
    <FILE>
    ghost/admin/lib/koenig-editor/addon/components/koenig-card-image
           /selector-tenor.js
    </FILE>
    <FILE>
    ghost/admin/lib/koenig-editor/addon/components/koenig-card-image
           /selector-tenor/gif.js
    </FILE>
</PATCHEDFILES>
</COMMIT>
</PATCHCOMMITS>
</ISSUE>
```

Figure 2: XML-based representation of a closed issue extracted from the Ghost project along with its associated metadata. The issue arises due to the use of a deprecated REST API endpoint that returns an HTTP 401 Unauthorized error.

3.2.2 *Classifying Issues into REST API and Non-REST API Defects.*

While the mined issues occur in REST-API-based projects, not all of them can be considered as **REST API defects**. We define *REST API defects* as *faults or failures that are directly related to the design, implementation, or behavior of the* ***RESTful interfaces*** *of a project, rather than other components.*

For example, the issue https://github.com/TryGhost/Ghost/issues/18903 from Ghost project does not qualify as a REST API defect. This issue arises when a user deletes the final character of a link within a Toggle card UI, which sometimes removes the entire link unexpectedly. The issue is related to how the link deletion feature is implemented in the frontend or editor plugin logic of the Ghost project. Furthermore, all the patched files used to fix this issue are unrelated to any REST API interface, which confirms that this issue is not a REST API defect.

LLM Prompt: You are an expert in software quality and REST APIs. A REST API defect refers to any observable failure or regression in the behavior of a RESTful system as visible to the client. This includes:
- Incorrect HTTP status codes
- Malformed, incomplete, or inconsistent JSON responses
- Missing required parameters or fields
- Violations of documented API contracts or expected behaviors
- Incorrect data in successful response that violates factual correctness
- incorrect request construction, missing parameter support, or regressions that break previously valid inputs
- Backward-incompatible changes in input handling or return values
- Unsuccessful client-server communication because all requests fail

Given the following GitHub issue title, description, and modified file types, determine whether it describes a REST API defect.
Respond *only* with a JSON object exactly matching this format, with no extra text or code blocks:
`{{"label":"bug" or "no-bug", "confidence":float between 0 and 1}}`
Title: {title}
Description: {desc}
Patched File Types: {patched_file_types}

Figure 3: Prompt template used for automatically classifying issues into REST API and non-REST API defects using LLM.

Automated issue classification: Instead of manually analyzing 11,313 issues to identify the REST API defects, we developed a semi-automated approach that uses large language model (LLM) to classify the mined issues into REST API defects and Non-REST API defects along with providing the confidence score ranging in $[0, 1]$. For this, multiple authors first manually and independently analyzed a set of randomly sampled 50 issues and classified them into REST-API and non-REST-API defects along with identifying the criteria they used to classify an issue as REST API defect. The analysis results were reconciled to engineer a prompt for LLM to automatically do the classification on the same set of 50 issues. The prompt was iteratively refined until LLM was able to accurately classify all the issues with at least 0.7 confidence score on the scale of $[0, 1]$. Figure 3 shows the final prompt crafted for classifying a closed issue into REST API (bug) and Non REST API defect (no-bug) categories. As shown, the prompt includes the eight criteria that were identified based on manual assessment of the REST API issues. Additionally, we found that while the information about developer patched files were helpful in classification, including absolute file paths or filenames in the prompt led to noise. Therefore, we classified the patched files into eight types based on the file extensions or file path patterns derived from the manual analysis of the patched files and used these types in the prompt. Table 1 lists these eight types and associated extensions and patterns.

For each issue, we extracted its *title*, *description*, and *patched files* from its XML representation, classified the patched files into unique file types, and crafted the prompt shown in Figure 3 by replacing the `{title}`, `{description}`, and `{patched_file_types}` with the extracted information. We then queried OpenAI's GPT-4.1-mini[3] LLM, which is part if the GPT-4.1 family of models to classify that issue as REST API defect or non-REST API defect along with providing the confidence score in range $[0, 1]$ for the classification. We selected GPT-4.1-mini because it balanced cost, speed, and accuracy for processing $11,313$ issues. Pilot tests with larger model (GPT-4.1)

[3]https://platform.openai.com/docs/models/gpt-4.1-mini

Table 1: Patched file types and associated extensions/patterns

Patched File Type	File Extensions / Filename Patterns
`test-file`	`_test.java`, `_test.py`, `spec.js`, `test.js`, contains "test"
`documentation-file`	`.md`, `.rst`, `.yaml`, `.json`, `.adoc`, `.markdown`, contains "readme" or "changelog"
`data-file`	`.json`, `.yaml`, `.yml`, `.xml`, `.csv`, `.po`, `.lang`, contains "data", "dataset", or "resources"
`config-file`	`.yml`, `.yaml`, `.json`, `.xml`, `.properties`, `.conf`, `.sh`, `.in`, `.j2`, `.toml`, `.sln`, `.csproj`, `.props`, `.cfg`, contains "mvn", "gradle", "pom.xml", "build.gradle", "docker-compose", "helm", "requirements", "makefile", "config", "go.mod", "go.sum", "yarn.lock", "nuget.config"
`source-file`	`.java`, `.py`, `.js`, `.jsx`, `.ts`, `.cpp`, `.c`, `.go`, `.rb`, `.cs`, `.h`, `.php`, `.proto`, `.tsx`
`database-file`	`.sql`, `.db`, `.sqlite`, `.psql`
`container-file`	contains "dockerfile"
`ui-file`	`.html`, `.css`, `.scss`, `.png`, `.svg`, `.woff`, `.less`, `.ttf`, `.eot`, `.cshtml`, `.vue`, `.hbs`
`other-file`	All others not matched above

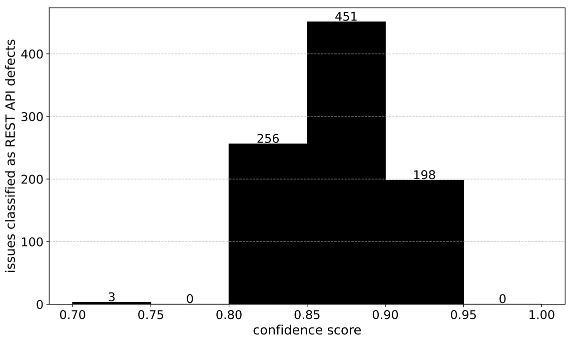

Figure 4: Distribution of 908 LLM-classified REST API defects (score ≥ 0.7) across confidence levels.

gave similar accuracy at higher cost, while smaller model (GPT-3.5) produced lower accuracy. Since all outputs were manually validated as described next, GPT-4.1-mini was an efficient, reliable choice. We considered the issues for which confidence score was at least 0.7 for further analysis, which resulted in selecting 908 out of the 11,313 issues. Figure 4 shows the histogram of these 908 issues automatically classified as REST API defects across different confidence scores ranging from 0.7 to 1.0. As shown, for most issues the LLM classified them as REST API defects with scores in the $[0.80, 0.95]$ range, indicating high certainty in classifying defects.

Manual validation: Using grounded theory analysis [5, 6, 18], two authors independently analyzed all 908 issues classified as REST API defects by the LLM and iteratively reconciled their analyses. Considering the eight criteria determined for automated classification, each author reviewed the issue's title, description, and patched files to determine if it constituted a true REST API defect. They eventually agreed on 852 issues, including 607 labeled as REST API defects by both, resulting in a raw agreement of 93.8% and a Cohen's kappa (κ) of 0.86, which indicates almost perfect agreement. Disagreements (56 issues) arose from continuous integration (CI)-generated issues or enhancement requests that resembled user-filed defects. These were discussed and reconciled through consensus.

The remaining 301 issues were labeled as false positives. For example, LLM classified the issue https://github.com/NatLibFi/Annif/issues/470 as REST API defect even though the bug lies in the neural network model that returns scores >1.0 via REST API, violating the intended range of $[0.0, 1.0]$. This issue had nothing to do with the REST API interface implementation. Overall, our analysis identified 607 REST API defects from 11,313 closed issues across 51 projects. Table 2 shows the distribution of manually curated 607 and LLM classified 908 REST API defects from the 11,313 closed issues across 51 projects along with project metadata including programming language, stars, forks, and lines of code (LOC). Since one project (*stf*) had no REST API defects, we focused on the remaining 50 projects to which the 607 defects belong for further analysis and benchmark construction. Due to reproducibility challenges with legacy dependencies [27], Defects4REST v1.0 implementation [14] provides set up to replicate 110 defects from 12 projects (highlighted in Table 2), with ongoing work to add more APIs and defects.

It is worth noting that automatically classifying issues is an active research area [11], where researchers have developed machine learning techniques to automatically classify and label issues. However, existing techniques cannot be used to classify issues into REST API and non-REST API defects because these techniques focus on classifying issues into labels such as *bug*, *enhancement*, and *question*. Thus, our LLM-based classification approach to identify REST API defects extends the state-of-the-art issue classification techniques.

3.3 Deriving a Taxonomy of REST API Defects

To derive a taxonomy of REST API defects from the dataset of 607 defects across 50 open-source projects, we develop a semi-automated methodology that combines clustering and topic modeling to guide and accelerate manual assessment. Our methodology consists of: (1) preprocessing issue reports to prepare them for clustering and topic modeling (Section 3.3.1), (2) clustering the preprocessed issue texts and performing topic modeling over the clusters (Section 3.3.2), and (3) manually assessing and labeling the derived topics to produce a final taxonomy of defect types and sub types (Section 3.3.3).

3.3.1 Preprocessing Issue Reports. To prepare the issue reports for topic modeling, we develop a comprehensive preprocessing pipeline that cleans and standardizes the textual content of each issue's title and description. We begin by concatenating the title and description of the issue reports followed by lowercasing the text and removing symbols, numbers, punctuation, and links including HTTP URLs and markdown-formatted references. We eliminate inline and block-style code snippets, file paths, and programming language identifiers to reduce noise. To improve token granularity, we split compound tokens written in camelCase, snake_case, or kebab-case. The resulting tokens are filtered to remove all English stopwords along with domain-specific and project-specific terms such as "fix", "merge", or API names like "seaweedfs" or "ghost". Finally, we discard very short tokens (e.g., one- or two-letter words) and reconstruct the cleaned tokens into a normalized text string for each issue. This preprocessing step ensures that our clustering and topic modeling uses semantically relevant terms rather than superficial or repetitive vocabulary.

3.3.2 Clustering and Topic Modeling. To identify coherent groups of related issues, we cluster the preprocessed issue texts using the DBSCAN algorithm following the prior work [13]. Each issue's text is first transformed into a dense vector representation using a

Table 2: Distribution of 607 manually-verified developer-fixed REST API defects across 51 open-source projects used to derive the REST API defect taxonomy. Highlighted rows: projects included in Defects4REST v1.0; *PRAB*: whether API is part of the Public REST API Benchmark [3]; *man*: 607 manually verified defects; *llm*: 908 LLM-classified defects from 11,313 closed issues.

Project	Language	PRAB	Description	#Defects (man/llm)	#Issues	#Stars	#Forks	#LOC
Annif	Python	✗	multilingual, open-source toolkit for automated subject indexing (i.e. tagging and classifying documents using machine learning)	3/7	120	212	142	5463
apistar	Python	✗	framework for building web APIs in Python, with support for type annotations and automatic schema generation	3/3	27	5571	412	25133
awx	Python	✗	web-based UI and API for managing Red Hat Ansible playbooks, inventories, and scheduled jobs	58/83	745	14323	3465	224784
Catwatch	Java	✓	web application that collects and exposes GitHub stats via a REST API to highlight project popularity and contributor activity	1/1	14	59	21	10647
cgm-remote-monitor	JavaScript	✗	web application providing a cloud-hosted interface for real-time upload, storage, and visualization of continuous glucose monitor data	4/4	76	2488	72058	155410
cwa-verification-server	Java	✓	application part of Germany's official COVID-19 exposure app, enabling secure verification of exposure data using Apple and Google's API	2/2	8	343	108	8068
digdag	Java	✓	tool to build, run, schedule, and monitor complex pipelines of tasks	2/4	59	1316	227	204600
djoser	Python	✗	application providing Django REST Framework endpoints for user operations like registration, login, and, password reset	10/11	42	2579	462	10797
dolibarr	PHP	✗	ERP (Enterprise Resource Planning) and CRM (Customer Relationship Management) software suite	29/35	482	5810	2892	2662024
DSpace	Java	✗	turnkey repository application for managing and providing access to digital content (e.g., scholarly publications and datasets)	19/26	197	927	1325	517595
elassandra	Java	✗	application integrating Elasticsearch with Apache Cassandra, allowing users to run Elasticsearch queries on data stored in Cassandra	10/10	50	1717	197	1329432
elasticsearch	Java	✗	distributed, scalable, and real-time search and analytics engine used for full-text search, log analytics, and real-time data exploration	9/55	912	71904	25113	4711791
enviroCar-server	Java	✓	manages users and anonymized XFCD OpenData, offering a RESTful API for accessing driving stats and spatiotemporal data	8/9	51	31	31	110451
flowable-engine	Java	✗	lightweight Java-based engine for executing BPMN, CMMN, and DMN business process models	21/29	207	8260	2680	1648598
Ghost	JavaScript	✗	headless CMS (Content Management System) designed for publishing blogs, newsletters, and online content	16/20	147	48.3K	10.5K	671933
granary	Python	✗	library and REST API for converting social network data to formats like ActivityStreams and JSON Feed for cross-platform interoperability	1/8	62	468	111	54458
harness	Go	✗	open-source codebase for the Harness platform, which is a continuous delivery (CD) and DevOps platform	35/58	1037	32484	2829	540543
hummingbot	Python	✗	software for building high-frequency crypto bots with standardized connectors to automate strategies across exchanges	2/2	13	11750	3293	244870
hydrus	Python	✗	REST API backend using Hydra Core Vocabulary to expose and manage data via a Hypermedia-driven Linked Data API	3/3	24	196	130	18200
kafka-rest	Java	✓	RESTful interface to Kafka, enabling data production, consumption, and cluster management without native Kafka clients	6/6	36	90	657	80543
label-studio	JavaScript	✗	data labeling tool for importing data, exporting annotations, managing projects, and integrating ML models	2/3	34	21056	2591	694057
localsend	Dart	✗	cross-platform app for secure file and message sharing between nearby devices over a local network via REST API and HTTPS	2/2	9	58517	3144	157443
management-api-for-apache-cassandra	Java	✓	sidecar service for centrally managing Apache Cassandra nodes, offering standardized operational actions via a Java agent	1/1	88	77	52	46981
mastodon	Ruby	✗	open-source, self-hosted decentralized social network server part of the Fediverse using Activity-Pub	16/21	146	47876	7119	888211
Mobile-Security-Framework-MobSF	JavaScript	✗	all-in-one mobile app security testing framework with static/dynamic/malware analysis and REST API for automation	5/6	104	18182	3328	709383
modular-monolith-with-ddd	C#	✗	full modular monolith app demonstrating DDD, clean architecture, CQRS, and REST APIs for module communication	1/2	9	11461	1799	49968
mygpo	Python	✗	backend for gpodder.net syncing podcast subscriptions, episode actions, and device data across devices	2/2	7	285	108	131354
netbox	Python	✗	open-source IRM tool for managing IP spaces, racks, devices, circuits, VMs, tenancy, and more using Django and PostgreSQL	9/24	140	17032	2659	727538
nocodb	TypeScript	✗	no-code platform turning any SQL database into a smart spreadsheet UI to build apps like CRMs or admin panels	47/63	465	51730	3581	2604120
nopCommerce	C#	✗	ASP.NET Core e-commerce platform with product catalog, cart, checkout, payments, shipping, and promotions	31/53	2617	9519	5452	852292
OrchardCore	C#	✗	modular, multi-tenant ASP.NET Core framework for building CMS, web apps, and modular systems	10/15	494	7.6K	2.4K	1229194
outline-server	TypeScript	✗	proxy server running Shadowsocks with a REST API for creating and managing access keys for VPN deployment	2/2	12	5896	796	32535
plane	TypeScript	✗	project management tool with APIs for tracking issues, managing sprints, and organizing product roadmaps	6/7	99	32352	1922	454275
podman	Go	✗	daemonless container engine for managing OCI containers, offering rootless mode, systemd support, and Docker compatibility	47/67	838	25601	2539	2080605
redash	Python	✗	open-source platform for querying databases, visualizing results, and building dashboards	19/46	391	27032	4434	150261
refugerestrooms	Ruby	✓	web app mapping safe restrooms for gender-diverse people with an API for location data access and visualization	1/1	9	913	264	16940
restcountries	Java	✓	provides detailed information about countries (like name, capital, population, currencies, languages, etc.)	16/16	26	2251	358	34443
seaweedfs	Go	✗	distributed file system for efficient, scalable object storage and backend use	40/58	391	23853	2357	418442
shopizer	Java	✗	headless e-commerce platform offering RESTful APIs for products, orders, customers, and more	2/3	25	3658	3068	97788
signal-cli-rest-api	Go	✗	Dockerized REST API for programmatic access to Signal Messenger via signal-cli	23/28	90	1522	177	19889
silver	Python	✗	Django-based billing system with a REST API for subscriptions and payment integration	7/11	57	304	101	28884
SpaceX-API	JavaScript	✓	REST API offering data on SpaceX launches, rockets, Starlink, and launchpads	6/7	18	10609	948	32777
spring-petclinic-rest	Java	✓	backend REST API for managing a pet clinic's pets, owners, visits, and vets	9/9	36	497	903	18165
stf	JavaScript	✗	web platform to control and test smartphones remotely using REST APIs	0/1	62	13471	2824	81266
strapi	TypeScript	✗	headless CMS for building APIs and managing content across frontends	5/7	31	65389	8381	505223
supabase	TypeScript	✗	backend-as-a-service with PostgreSQL, auth, storage, and auto-generated REST API	1/1	68	78552	7893	2233054
traefik	Go	✗	reverse proxy and load balancer with REST APIs for dynamic routing and monitoring	1/1	5	53561	5253	387324
uptime-kuma	JavaScript	✗	self-hosted service monitor with REST APIs for metrics, status pages, and more	4/10	126	65521	5781	139890
vercel	TypeScript	✗	cloud platform for frontend developers to deploy sites and serverless functions	11/26	478	13321	2419	2141719
WP-API	PHP	✗	REST API for WordPress to access posts, pages, users, and other content	37/37	115	3942	655	23592
zuul	Java	✗	API gateway providing routing, monitoring, and security for REST API traffic	2/2	14	13644	2405	29375
Total	—	—	—	**607/908**	**11,313**	—	—	—

pre-trained Sentence-BERT model (all-MiniLM-L6-v2) [21]. These embeddings capture the semantic meaning of the issue reports. DBSCAN is then applied with a cosine distance metric and a tunable epsilon threshold to group similar issue reports while filtering out noise and small, incoherent clusters. Clusters with fewer than a specified minimum number of documents are discarded to ensure topic stability. This filtering step results in a reduced set of high-quality, semantically similar issue reports, along with their corresponding embeddings, issue URLs, and patched file types.

Topic modeling: After identifying high-quality clusters of issue reports using DBSCAN, we retain the issues from dense regions and apply BERTopic [9] to extract interpretable topic representations. BERTopic is a topic modeling technique that uses class–based TF-IDF (c-TF-IDF) to generate concise topic descriptions by identifying terms that are statistically salient across clusters of semantically similar documents (issues). To control the vocabulary used for topic representation, we initialize BERTopic with a custom `CountVectorizer` from scikit-learn [20], configured to extract unigrams and bigrams while filtering out overly rare or common terms using minimum and maximum document frequency thresholds. This setup produces more meaningful and distinctive topic representations, facilitating better interpretation of the underlying topics or defect categories.

Hyper-parameter tuning: We perform hyper-parameter tuning using grid search over a comprehensive set of configurations as summarized in Table 3. This resulted in 1, 800 distinct configurations evaluated to identify the optimal clustering parameters. For each configuration, we compute three key evaluation metrics to assess clustering quality: (1) **Topic Coherence** [22]: Measures the semantic similarity among the top words within each cluster, providing an indication of how interpretable and meaningful the discovered topics are. For noisy and diverse textual data such as issue reports, a coherence score in the range $[0.35, 0.5]$ is generally considered acceptable, reflecting moderately interpretable topics despite data variability. Scores > 0.5 typically indicate well-formed and highly interpretable topics. Conversely, lower scores suggest that clusters are too generic or contain a mixture of unrelated issues, reducing their practical usefulness for understanding defect types. (2) **Silhouette Score** [23]: Quantifies how well-separated the clusters are by measuring the similarity of an issue to its own cluster compared to other clusters. This ranges from -1 to 1, and values >0.7 indicate strong clusters. In the context of issue reports, silhouette scores often tend to be lower ($(0, 0.1]$) because most issue reports contain information relevant to multiple defect types, resulting in overlapping clusters and less distinct boundaries. As a result, low silhouette scores in this setting do not necessarily indicate poor clustering quality, but rather reflect the inherent ambiguity and multi-label nature of the data. (3) **Total Topics**: The total number of clusters (topics) identified for a given configuration, reflecting the granularity of the clustering. Having more topics indicates a finer-grained clustering that may capture subtle distinctions between defect types, but could also lead to overly fragmented or redundant topics. In contrast, fewer topics reflects a coarser clustering, which may group together broader categories of defects but risks overlooking important nuances or specific defect types present in the data. The optimal number of topics depends on the diversity of the issue reports.

Table 3: Hyperparameter configurations explored for clustering and topic modeling.

Parameter	Description	Values Explored
eps(DBSCAN)	Maximum distance between points in the same neighborhood. Controls density threshold for clustering.	0.5, 0.6, 0.7
`min_samples` (DBSCAN)	Minimum number of neighboring points required to form a dense region (core point).	10, 20, 30
`min_cluster_size` (Post-DBSCAN filter)	Minimum number of documents in a cluster to retain it for topic modeling. Clusters smaller than this are discarded.	10, 20, 30, 40, 50
`min_df` (CountVectorizer)	Minimum number of documents a term must appear in to be included in the topic vocabulary. Filters out rare words.	2, 3, 4, 5
`max_df` (CountVectorizer)	Maximum number of documents a term can appear in before being excluded. Filters overly common words.	300, 400, 500, 600, 700
`ngram_range` (CountVectorizer)	Size of n-grams extracted for topic representation.	(1, 2)

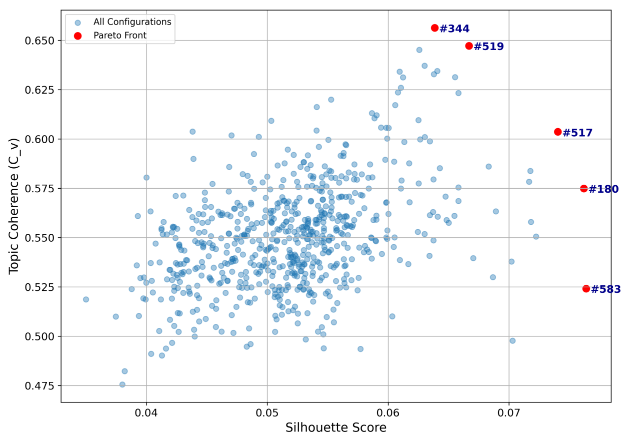

Figure 5: Pareto front of DBSCAN-BERTopic configurations showing the tradeoff between topic quality (Topic Coherence) and cluster separation (Silhouette Score). The five annotated points represent configurations with $\geq$ 10 topics that offer the best balance between topic quality and cluster separation.

To empirically determine optimal metrics, we conducted a grid search over 1, 800 parameter configurations, of which 864 produced between 2 and 15 topics. DBSCAN's key parameters—how semantically related the top terms in each topic are (*topic coherence*) and how well-separated the topics are (*silhouette score*)—control the density variation across clusters/topics, ensuring that both dense and sparse regions are appropriately captured. Configurations with <10 topics merged distinct topics, so we focused on 659 configurations with ≥10 topics. Next, we analyzed the trade-off between Topic Coherence and Silhouette Score by plotting these 659 configurations in a two-dimensional space. Configurations that lie on the *Pareto front*, meaning that no other configuration achieves both higher coherence and higher silhouette score, are considered optimal; this selection ensures that the final clusters are robust across variations in density. Figure 5 illustrates this Pareto front, showing five configurations that define the frontier. Each of these configurations resulted in 10–12 topics. We manually analyzed them to come up with a unique set of topics as described next.

3.3.3 Manual Topic Assessment. For each optimal configuration, we manually analyzed the derived topics that were labeled with their top-ranked terms and visualizations (topic-term rankings, inter-topic distances, and hierarchies) for validating the coherence and distinctiveness of the topics. We found that the topics derived across configurations were highly overlapping however, some configurations lead to distinct topics. We reconciled all the topics derived from the five configurations by merging overlapping ones and filtering out noisy clusters to create an interpretable taxonomy of 6 REST API defect types categorized into 13 sub defect types.

To assess how consistently defect types emerge across different topic modeling configurations, we analyzed their presence across the five optimal configurations, each defined by distinct hyperparameter settings. Table 4 shows these defect types along with the number of configurations using which they were derived and the number of defects associated with each defect type and sub defect type. As shown in Table 4, **Configuration and Environment Issues**, **Data Validation and Query Processing Errors**, **Integration, Middleware, and Runtime Environment Failures**, and **Data Storage, Access, and Volume Errors** types were observed in all five configurations, indicating that these issues are systemic and consistently discovered regardless of the clustering/topic modeling configuration parameters. **Authentication, Authorization, and Session Management Issues** appeared in four configurations underscoring their foundational importance in REST API robustness. In contrast, **Distributed Systems and Clustering Failures** occurred in two configurations, suggesting these issues are prevalent but more concentrated in specific systems. These findings suggest that defects related to **data handling**, **authentication and authorization**, and **request validation** are not only common but also generalizable across clustering/modeling configurations, reinforcing their occurrence as key REST API defect types. Although issues exhibited characteristics of multiple subtypes, for taxonomy construction we assigned each issue to its *dominant subtype*—the primary manifestation observed in the issue report and its corresponding patch. This approach is standard in taxonomy construction, where categories represent the most characteristic behavior of each instance. We manually reviewed and reconciled the resulting subtypes to ensure they were coherent and representative.

4 EVALUATION

This section describes our results and analysis in terms of the three research questions we ask in Section 1.

4.1 RQ1: Common REST API Defect Types

To address this research question, we analyze the distribution of manually verified 607 REST API defects across the top-level defect types as shown in Table 5. Among the six defect types, **Data Validation and Query Processing Errors** were the most frequent, accounting for 253 (42%) defects. These defects include issues such as schema and payload validation failures in POST APIs, and errors in query filters and search parameter handling. **Configuration and Environment Issues** followed with 164 (27%) defects, encompassing container and resource quota misconfigurations, job execution errors, and environment-specific configuration bugs. Both **Authentication, Authorization, and Session Management Issues** and **Data Storage, Access, and Volume Errors** were equally frequent, each comprising 62 (10%) defects. These categories include token handling errors, session lifecycle management, file upload problems, and database access issues. **Integration, Middleware, and Runtime Environment Failures** were also prominent, with 61 (10%) defects involving middleware integration issues, runtime errors, and process signal problems in containerized environments. Finally, **Distributed Systems and Clustering Failures** were the least common, with only 5 (1%) defects, primarily related to index coordination and cluster management failures. These results highlight that a large proportion of REST API defects relate to validation and configuration-related errors in real-world REST API systems.

RA1: Real-world REST APIs commonly suffer from data validation and query processing errors (42%), followed by configuration and environment-related problems (27%). Additional defect types include authentication and session management issues, data storage errors, and integration or runtime failures (each 10%). Defects related to distributed systems and clustering occur less often (1%).

4.2 RQ2: File Types and Resolution Time

To determine the types of files modified to fix REST API defects, we analyzed the distribution of developer-patched file types (listed in Table 1) across sub defect types (ST1 to ST13 listed in Table 4), as shown in Figure 6. Our analysis revealed that, developers most commonly modify *source code files* across all defect types, often alongside *test files*, *configuration files*, and *documentation*. For example, fixing *Schema and Payload Validation Errors in POST APIs* (253 files) frequently involves changes to source files (126), test files (46), and data files (22). Similarly, *Query Filter and Search Parameter Handling Errors* (222 files) often require modifications to source (88), test (44), and documentation files (28). In *Environment-Specific Behavior and Configuration Bugs* (228 files), source files (110) and configuration files (19) dominate. Some defect types such as *Authentication and Token Management Errors* and *Session Lifecycle Issues* also involve UI and data files, reflecting the cross-cutting impact of these bugs. Even low-frequency categories like *Index and Cluster Coordination Failures* (10 files) and *Containerized API Signal Issues* (5 files) show a consistent reliance on source and test file updates. Overall, the results suggest that while source code changes are central to most fixes, supporting files such as configuration, test, documentation, and UI components are also frequently modified, indicating the heterogeneous nature of REST API defects.

To understand how long it takes developers to resolve REST API defects, we analyzed the fix times (in days) across 13 sub types in our taxonomy as shown in Table 6. Prior work shows that *time to fix* reflects an issue's *priority* rather than actual effort [19]. Our analysis reveals that REST API defect types vary significantly in how urgently developers address them. For example, *Container and Resource Quota Handling Errors* (ST1) and *Database/Table User Access Handling Errors* (ST12) have the shortest average time to fix (13 and 22 days, respectively), suggesting that these issues may be considered critical or blocking in nature. In contrast, defects such as *Index and Cluster Coordination Failures* (ST13) (399 days) and *Session, Token, and Account Lifecycle Management Errors* (176 days) are resolved far more slowly on average, indicating either lower priority or greater delay in triage and resolution. Interestingly, commonly

Table 4: REST API defect taxonomy created by analyzing the topics derived from optimal DBSCAN-BERTopic configurations. "#Defects" shows the number of issues classified into each defect/sub defect type and "#Configs" shows how many times each sub defect type was observed across the topics derived using five optimal DBSCAN-BERTopic configurations.

Defect Type	Sub Defect Type	#Defects	#Configs	Description
Configuration and Environment Issues (T1)	Container and Resource Quota Handling Errors (ST1)	9	5	Failures in configuring or enforcing container limits and runtime settings.
	Job Execution and Workflow Configuration Defects (ST2)	25	5	Jobs or workflows fail due to incorrect API inputs or mis-configured templates.
	Environment-Specific Behavior and Configuration Bugs (ST3)	130	5	API behavior changes or breaks under specific environments or configurations.
Data Validation and Query Processing Errors (T2)	Schema and Payload Validation Errors in POST APIs (ST4)	135	5	API requests are rejected due to malformed JSON, invalid fields, or schema violations.
	Query Filter and Search Parameter Handling Errors (ST5)	118	4	Query or search parameters are malformed, unsupported, or cause unexpected results.
Authentication, Authorization, and Session Management Issues (T3)	Authentication and Token Management Errors (ST6)	34	4	User authentication or token-based access fails due to mis-configured endpoints.
	Session, Token, and Account Lifecycle Management Errors (ST7)	28	1	Problems with login sessions, token refresh, or account-related workflows.
Integration, Middleware, and Runtime Environment Failures (T4)	Middleware Integration Failures in REST APIs (ST8)	39	5	Failures during REST API interaction with CI tools, middleware, or web servers.
	Process Signal and Grouping Issues in Containerized APIs (ST9)	3	5	Containerized services fail to handle process signals or groups correctly.
	Runtime and Dependency Errors (ST10)	19	4	APIs fail due to packages/framework, environment, or import-related issues.
Data Storage, Access, and Volume Errors (T5)	Volume and File Upload/Access Errors (ST11)	42	5	File or volume operations fail due to missing paths, upload errors, or HTTP issues.
	Database/Table User Access Handling Errors (ST12)	20	5	Users face access issues due to database permission errors or missing table configurations.
Distributed Systems and Clustering Failures (T6)	Index and Cluster Coordination Failures (ST13)	5	2	Distributed systems fail due to index mismatches, sync issues, or type conflicts.

Table 5: Top REST API Defect Types Ranked by Frequency

Rank	Defect Type	#Defects
1	Data Validation and Query Processing Errors	253
2	Configuration and Environment Issues	164
3	Authentication, Authorization, and Session Management Issues	62
3	Data Storage, Access, and Volume Errors	62
4	Integration, Middleware, and Runtime Environment Failures	61
5	Distributed Systems and Clustering Failures	5
total		607

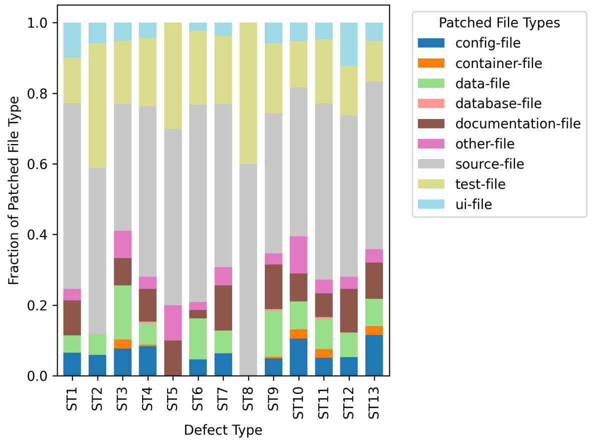

Figure 6: Distribution of patched file types per sub defect type listed in Table 4.

occurring defect types such as *Schema and Payload Validation Errors in POST APIs* (ST4) and *Query Filter and Search Parameter Handling Errors* (ST5) are fixed relatively faster on average (65 and 99 days, respectively), implying higher responsiveness due to their functional impact. These findings highlight the varying prioritization of defect types in REST APIs and suggest opportunities for more systematic triaging of issues that may otherwise linger despite their technical significance.

Table 6: Statistics of fix times (in days) per defect and sub-defect type listed in Table 4.

Defect Type	Sub Defect Type	Min	Max	Mean	Std Dev
T1	ST1	0	58	13.33	19.51
T1	ST2	0	189	42.64	64.63
T1	ST3	0	1690	88.19	247.77
T2	ST4	0	1451	64.72	173.69
T2	ST5	0	1185	99.48	202.02
T3	ST6	0	1141	103.12	226.32
T3	ST7	0	2788	175.82	527.48
T4	ST8	0	1684	93.46	292.22
T4	ST9	0	98	34.33	55.19
T4	ST10	0	988	90.05	226.41
T5	ST11	0	553	49.60	101.39
T5	ST12	0	235	21.75	52.81
T6	ST13	6	1672	398.80	718.64

RA2: Source files are most frequently modified across all REST API defect types alongside configuration, test, documentation, and UI files, indicating their heterogeneous nature. Fix times vary significantly across defect types, with some types fixed within a few weeks while others remain open for years.

4.3 RQ3: REST API Testing Tool Evaluation

We examine how real-world REST API defects correspond to faults detected by four state-of-the-art REST API testing tools through qualitative (Section 4.3.1) and quantitative (Section 4.3.2) analyses.

4.3.1 Qualitative Analysis. We first mapped our 13 defect sub types to the fault taxonomy from [13], which categorizes defects detected by EvoMaster [1] across seven open-source Java APIs. Table 7 shows this mapping, revealing both strengths and limitations of EvoMaster, which is an evolutionary search-based API testing tool.

Strengths: As shown, EvoMaster effectively detects defects related to input validation and schema conformance. Defects such as *Schema and Payload Validation Errors in POST APIs* and *Query Filter and Search Parameter Handling Errors* are well aligned with its capabilities, which leverage OpenAPI-based input generation, status code monitoring, and response structure validation.

Limitations: However, many defect types are only partially mapped or missed entirely. For example, *Session, Token, and Account*

Table 7: Mapping REST API sub defect types to faults detectable by EvoMaster; Unmapped and Partially mapped denote defects that are difficult or impossible to detect.

Sub Defect Type	Mapped Fault Type (EvoMaster taxonomy) [13]
Container and Resource Quota Handling Errors	**Unmapped** (Requires container/runtime configuration introspection; beyond EvoMaster's scope)
Job Execution and Workflow Configuration Defects	**Partially mapped:** Inconsistent Behavior (May detect if failures result in bad HTTP responses; but not misconfigured workflows)
Environment-Specific Behavior and Configuration Bugs	**Partially mapped:** Framework Misconfiguration, Inconsistent Behavior (EvoMaster doesn't support cross-environment execution)
Schema and Payload Validation Errors in POST APIs	Schema Conflicts
Query Filter and Search Parameter Handling Errors	Schema Conflicts
Authentication and Token Management Errors	**Partially mapped:** Faults in the Business Logic (May work if auth is stubbed or hardcoded; fails if real token exchange is needed)
Session, Token, and Account Lifecycle Management Errors	**Unmapped** (Requires managing long-term state and valid accounts)
Middleware Integration Failures in REST APIs	**Unmapped:** (External system failures (e.g., CI, middleware) aren't visible to EvoMaster)
Process Signal and Grouping Issues in Containerized APIs	**Unmapped** (EvoMaster cannot simulate or detect OS-level process issues)
Runtime and Dependency Errors	Configuration and/or Execution Faults – Inconsistent Behavior
Volume and File Upload/Access Errors	**Unmapped**
Database/Table User Access Handling Errors	**Partially mapped:** Data Integrity Faults – DB Injection; DB Operations
Index and Cluster Coordination Failures	**Unmapped** (EvoMaster works on a single-instance API; cluster coordination bugs are out-of-scope)

Lifecycle Management Errors (e.g., mastodon#30103) and *Middleware Integration Failures* (e.g., dolibarr#32072) often involve subtle state transitions, multi-step workflows, or interactions with external systems. These are challenging to detect due to limited semantic understanding and support for long-lived sessions or middleware responses. Similarly, *Container and Resource Quota Handling Errors* (e.g., awx#4910) and *Volume and File Upload/Access Errors* (e.g., seaweedfs#913) depend on runtime conditions or system configurations, which lie outside EvoMaster's scope.

4.3.2 Quantitative Analysis. To validate and extend our qualitative findings, we evaluated three state-of-the-art REST API testing tools besides EvoMaster on a stratified sample of 30 defects covering all 6 defect types and 13 sub defect types across 11 projects.

Selection criteria: We selected testing tools considering four diverse paradigms: schema-based property testing (Schemathesis [10]), model-based fuzzing (RESTler [2]), learning-based test synthesis (AutoRestTest [25]), and evolutionary search-based test synthesis (EvoMaster [1]). We constructed a defect dataset using stratified sampling across the taxonomy, selecting 2–3 defects per subtype for each of the 13 subtypes while ensuring no single project was over-represented. This resulted in a subset of 30 defects from 11 projects covering all 6 defect types with their 13 subtypes.

Experiment procedure: We configured each tool according to its documentation with default settings and provided authentication credentials where required using API keys or bearer tokens specified in OpenAPI specifications. Following recent study [25], we executed all four tools with 1 hour timeout per-seed using 5 random seeds for each of the 30 defects. This required 30 defects × 4 tools × 5 seeds = 600 tool executions with 1 hour per execution.

To determine whether a tool detected a defect, we analyzed the generated tests and execution logs to check if the tool invoked the defect-triggering endpoint(s) and produced an error that matched or closely resembled the incorrect behavior. We performed this analysis for each of the five seeds' results and if a tool detected the defect even in one seed, we conservatively concluded that the tool detected it. For the defects not detected, we classified them into *potentially detect* and *cannot detect* categories considering tools' capabilities and those required to detect those defects. We provide detailed analysis for each defect–tool pair in our replication package along with the code to execute tools, generated tests, and logs.

Table 8: The effectiveness of four state-of-the-art REST API testing tools (Schemathesis (S) [10], RESTler (R) [2], EvoMaster (E) [1], and AutoRestTest (A) [25]) in detecting 30 real-world defects from 11 APIs, covering 6 defect types and 13 subtypes. ✓: defect detected; ✗: tool cannot detect the defect; ●: tool could potentially detect the defect; ✠: tool crashed or terminated unexpectedly

Sub Defect Type	Defect ID	API#GitHub Issue No.	S	R	E	A
Container and Resource Quota Handling Errors (ST1)	D1	podman#14676	✗	✗	✗	✗
	D2	seaweedfs#913	●	●	●	✓
Job Execution and Workflow Configuration Defects (ST2)	D3	flowable-engine#2584	✓	✓	✗	✗
	D4	dolibarr#30950	✗	✗	✗	✗
Environment/Configuration Bugs (ST3)	D5	restcountries#201	✗	✗	✗	✗
	D6	dolibarr#26307	●	●	✠	✠
Schema/Payload Validation Errors in POST APIs (ST4)	D7	podman#23981	●	●	●	✠
	D8	dolibarr#33949	✗	✗	✗	✗
Query Filter and Search Parameter Handling Errors (ST5)	D9	restcountries#235	✗	✗	✗	✗
	D10	signal-cli-rest-api#654	✗	✗	✗	✗
	D11	awx#9222	✗	✗	✗	✗
Authentication and Token Management Errors (ST6)	D12	dolibarr#26066	✗	●	●	✗
	D13	awx#7243	✗	●	●	✗
Session, Token, and Account Management Errors (ST7)	D14	mastodon#30103	✗	✗	✗	✗
	D15	enviroCar-server#45	●	●	✗	●
Middleware Integration Failures in REST APIs (ST8)	D16	flowable-engine#3856	✗	✗	✗	✗
	D17	mastodon#30039	✗	✗	✗	✗
	D18	dolibarr#32072	✗	✗	✗	✗
Process Signal and Grouping Issues in Containerized APIs (ST9)	D19	podman#19368	●	●	✗	●
	D20	signal-cli-rest-api–387	✗	✗	✗	✗
	D21	podman#18424	✗	✗	✗	✗
Runtime and Dependency Errors (ST10)	D22	nocodb#2776	●	✠	✓	✗
	D23	signal-cli-rest-api–103	✗	✗	✗	✗
Volume and File Upload/Access Errors (ST11)	D24	podman#15720	✗	✗	✗	✗
	D25	flowable-engine–3003	✗	✗	✗	✗
	D26	seaweedfs#5864	✗	✗	✗	✗
Database/Table User Access Handling Errors (ST12)	D27	dolibarr#29372	●	●	●	✠
	D28	nocodb#7535	✗	✗	✗	✗
Index and Cluster Coordination Failures (ST13)	D29	kafka-rest–341	●	●	●	●
	D30	seaweedfs#5213	●	●	●	●
total	**30**		1	1	1	1

Findings: Table 8 summarizes the detection results. Overall, the four tools detected only **3 out of 30 defects (10%)**: AutoRestTest detected D2 (Container and Resource Quota handing error), Schemathesis and RESTler both detected D3 (Job Execution workflow issue), and EvoMaster detected D22 (Runtime dependency error). EvoMaster and RESTler crashed on one defect while AutoRestTest crashed on 6 of the 11 APIs. Such crashes highlight the fragility of existing schema-driven testing approaches when applied to real-world APIs with nonstandard specifications, authentication requirements, or environment-dependent schemas. Tools could **potentially detect 9 (30%) additional defects** (D6, D7, D12, D13, D15, D19, D27, D29, D30) given longer execution time, better input generation, or complete OpenAPI specifications, but fundamentally **cannot detect 18 (60%) defects** (D1, D4, D5, D8–D11, D14, D16–18, D20, D21, D23–26, D28) due to inherent limitations. Detection rates varied significantly across subtypes: ST1–ST4, ST6, ST7, ST9, ST10, ST12, and ST13 showed 1 or 2 detections/potential detections, while ST5, ST8, and ST11 had 0 detections each.

Root cause analysis: Our analysis revealed nine fundamental limitations preventing testing tools from detecting the 18 defects:
L1. Semantic Correctness vs. Schema Conformance (8/18 = 44.4% defects). All tools validate schema but not semantic correctness of returned values (D4: incorrect reference suffix; D5: wrong timezone; API fails to persist PUT/POST value (D8, D16, D18); D9: incorrect capital name; D24: incorrect `RefCount` for volumes; D25 :incorrect media type for .form files).
L2. Stateless Testing (2/18 = 11.1% defects). EvoMaster and AutoRestTest cannot follow deep multi-step workflows requiring state propagation (D1: create, start, and query memory stats of a container; D14: create and push notification subscriptions using multiple users).
L3. Authentication Gaps (2/18 = 11.1% defects). Schemathesis and AutoRestTest attach API keys to all requests; RESTler and EvoMaster require explicit 401 specifications. These tools cannot emulate multiple users with different credentials (D28, D14).
L4. Infrastructure Context (1/18 = 5.6% defects). Defects tied to containers, volumes, or clusters require environment-specific behavior that tools cannot simulate (D26: WebDAV backend process).
L5. Incomplete OpenAPI Specifications (3/18 = 16.7% defects). Missing response codes (D12: no 401), custom filters (D11: Django `iexact`), permission model (D28: cannot access email notification), or inaccurate constraints make defects untestable.
L6. Business Logic Violations (1/18 = 5.6% defects). Tools miss logic faults masked by valid HTTP 200 responses (e.g., D17: idempotency violation creating duplicate posts).
L7. Integration-Level Behavior (2/18 = 11.1% defects). Some defects manifest only through internal component interactions (D10: `notify_self` messaging logic on Signal App and D16: query historic instances of a process created using specific variable). Black-box testing cannot exercise these defects.
L8. External Dependencies (2/18 = 11.1% defects). Require valid identifiers or accounts that existing tools cannot generate (D20: registered phone numbers; D23: active Signal accounts).
L9. Time-Dependent Issues (1/18 = 5.6% defects). Tools ignore performance or runtime anomalies (D21: 5-min stale exec status).

RA3: REST API testing tools detect only 10% (3/30) but cannot detect 60% (18/30) of real-world defects. They can handle basic schema validation but miss authentication, infrastructure, session-management, semantic-correctness, and business-logic faults.

5 DISCUSSION

Our dataset of 607 real-world REST API defects across 50 open-source projects reveals critical insights for both researchers and practitioners. Defect type distribution shows 42% involve data validation and query processing, while resolution patterns reveal infrastructure issues are fixed quickly (2–3 weeks) compared to cluster coordination and session management (months to years), informing testing priorities and resource allocation. Our defect taxonomy spans authentication, session management, container/resource quotas, schema validation, storage access, and runtime configuration—many overlooked by current tools. Tool evaluation demonstrates existing approaches detect only 10% of real-world defects, revealing fundamental limitations that must be addressed to advance REST API testing and debugging research.

Implications for Practitioners and Future REST API Research: *For Practitioners:* (1) Complement automated API testing with manual tests for authentication workflows, infrastructure configurations, and business logic validations. (2) Ensure complete OpenAPI specifications with all response codes, authentication schemes, and constraints. (3) Implement stateful integration tests exercising multi-step workflows. (4) Incorporate infrastructure testing as API testing tools lack that capability.
For Researchers: (1) *Semantic-aware test generation:* Develop techniques that validate semantic correctness of API responses beyond schema conformance, potentially using API documentation or domain-specific invariants. (2) *Stateful testing:* Model multi-step workflows maintaining session state and data dependencies. (3) *Authentication testing:* Build tools that systematically test authentication and authorization by varying credentials, roles, and security contexts, including deliberately invalid authentication scenarios. (4) *Infrastructure-aware testing:* Integrate infrastructure concerns (containers, volumes, quotas) into test generation through combined API and infrastructure-as-code testing or deployment simulation. (5) *Specification mining:* Develop techniques to automatically infer missing constraints, response specifications, and framework-specific extensions from code, documentation, or runtime traces. (6) *Business logic inference:* Explore program analysis and learning-based approaches to infer business-logic invariants serving as test oracles beyond HTTP-level validation. (7) *Performance testing:* Extend tools to detect response time anomalies, hangs, and time-dependent inconsistencies. (8) *Fault localization (FL) and repair:* Beyond testing, use Defects4REST for developing and evaluating FL [17] and program repair [15, 16] techniques for REST APIs, addressing challenges of localizing faults across API endpoints, handlers, and database interactions, and automatically generating patches preserving API contracts and backward compatibility.

Threats to validity: *Construct validity* may be affected by annotation interpretation while *Internal validity* by repository selection bias (mitigated through diverse selection); *External validity* is limited to selecting open-source projects (proprietary APIs may differ); *manual validation* introduces subjectivity (minimized through dual annotation and high agreement: $\kappa = 0.86$, 93.8%). We release all code, annotations, and data to promote transparency.

6 CONCLUSION

We presented Defects4REST, the first comprehensive benchmark of real-world REST API defects, organized into a taxonomy of 6 types and 13 sub defect types. Our evaluation of four state-of-the-art testing tools reveals that current approaches detect only 10% of real-world defects, missing critical issues in authentication, infrastructure, semantic correctness, and business logic. Defects4REST provides a foundation for developing and evaluating more effective REST API testing, debugging, and repair techniques.

ACKNOWLEDGMENTS

We thank Kausar Y. Moshood and Huwaida R. Yafie for their assistance in setting up and replicating benchmark defects, and manually analyzing API testing tools' results. This research was supported by startup funds provided by Oregon State University.

REFERENCES

[1] Andrea Arcuri, Man Zhang, Amid Golmohammadi, Asma Belhadi, Juan P Galeotti, Bogdan Marculescu, and Susruthan Seran. 2023. Emb: A curated corpus of web/enterprise applications and library support for software testing research. In *IEEE Conference on Software Testing, Verification and Validation (ICST)*. 433–442.

[2] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 748–758. https://doi.org/10.1109/ICSE.2019.00083

[3] Alix Decrop, Sara Eraso, Xavier Devroey, and Gilles Perrouin. 2025. A Public Benchmark of REST APIs. In *IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 421–433.

[4] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures.* University of California, Irvine.

[5] Smita Ghaisas, Manish Motwani, Preethu Rose ANISH, and Shashi Kant Sharma. 2018. Automated classification of Business rules from text. US Patent 10,146,762.

[6] Smita Ghaisas, Manish Motwani, Balaji Balasubramaniam, Anjali Gajendragadkar, Rahul Kelkar, and Harrick Vin. 2015. Towards automating the security compliance value chain. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 1014–1017. https://doi.org/10.1145/2786805.2804435

[7] GitHub, Inc. 2022. *GitHub REST API Documentation.* GitHub. https://docs.github.com/en/rest?apiVersion=2022-11-28 API Version: 2022-11-28.

[8] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–41.

[9] Maarten Grootendorst. 2022. BERTopic: Neural topic modeling with a class-based TF-IDF procedure. *arXiv preprint arXiv:2203.05794* (2022).

[10] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web API schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 345–346. https://doi.org/10.1145/3510454.3528637

[11] Jueun Heo, Gibeom Kwon, Changwon Kwak, and Seonah Lee. 2024. A Comparison of Pretrained Models for Classifying Issue Reports. *IEEE Access* 12 (2024), 79568–79584. https://doi.org/10.1109/ACCESS.2024.3408688

[12] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[13] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the faults found in REST APIs by automated test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–43.

[14] Rahil P. Mehta, Pushpak Katkhede, and Manish Motwani. 2025. Defects4REST: A Benchmark of Real-World Defects to Enable Controlled Testing and Debugging Studies for REST APIs. GitHub repository. https://github.com/ANSWER-OSU/Defects4REST

[15] Manish Motwani. 2021. High-Quality Automated Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 309–314. https://doi.org/10.1109/ICSE-Companion52605.2021.00134

[16] Manish Motwani. 2022. High-Quality Automatic Program Repair. *UMass Dissertation* (2022).

[17] Manish Motwani and Yuriy Brun. 2023. Better Automatic Program Repair by Using Bug Reports and Tests Together. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1225–1237. https://doi.org/10.1109/ICSE48619.2023.00109

[18] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs?. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 25. https://doi.org/10.1145/3180155.3182533

[19] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering* 23, 5 (2018), 2901–2947. https://doi.org/10.1007/s10664-017-9550-0

[20] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[21] Fabian C Peña and Steffen Herbold. 2025. Evaluating the Performance and Efficiency of Sentence-BERT for Code Comment Classification. In *2025 IEEE/ACM International Workshop on Natural Language-Based Software Engineering (NLBSE)*. IEEE, 21–24.

[22] Michael Röder, Andreas Both, and Alexander Hinneburg. 2015. Exploring the Space of Topic Coherence Measures. In *ACM International Conference on Web Search and Data Mining* (Shanghai, China) *(WSDM '15)*. Association for Computing Machinery, New York, NY, USA, 399–408. https://doi.org/10.1145/2684822.2685324

[23] Peter J Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65.

[24] Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. 2023. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624* (2023).

[25] Tyler Stennett, Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2025. AutoRestTest: A Tool for Automated REST API Testing Using LLMs and MARL. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 21–24. https://doi.org/10.1109/ICSE-Companion66252.2025.00015

[26] Hao-Nan Zhu, Robert M Furth, Michael Pradel, and Cindy Rubio-González. 2025. From Bugs to Benchmarks: A Comprehensive Survey of Software Defect Datasets. *arXiv preprint arXiv:2504.17977* (2025).

[27] Hao-Nan Zhu and Cindy Rubio-González. 2023. On the Reproducibility of Software Defect Datasets *(ICSE '23)*. IEEE Press, 2324–2335. https://doi.org/10.1109/ICSE48619.2023.00195