

A Tool for Functional Testing of REST APIs

Diptikalyan Saha
diptsaha@in.ibm.com
IBM Research, India

Devika Sondhi
devika.sondhi@ibm.com
IBM Research, India

Swagatam Haldar
swagatam.haldar@student.uni-
tuebingen.de
University of Tübingen, Germany

Margaret Mary Dhanaswamy
margaret.d@in.ibm.com
IBM Consulting, Bangalore, India

Aman Goyal
aman.goyal1@ibm.com
IBM Consulting, Bangalore, India

Saritha Route
saritha.route@in.ibm.com
IBM Consulting, Bangalore, India

ABSTRACT

The advancement of cloud services has sparked renewed interest in testing REST APIs. Numerous testing techniques and tools have been developed to explore API endpoints using different strategies automatically. Their goal is to enhance coverage and identify bugs. However, these techniques can generate test cases that may not be suitable for creating functional test suites. This paper introduces a tool designed for automated functional testing of REST APIs. The tool is built on a novel algorithm capable of generating functional test cases. It automatically generates various functional sequences, parameter scenarios, data scenarios, and multiple oracle checks, and offers great flexibility and guidance for modifying the test cases. This allows users to refine the auto-generated test cases according to their application needs based on application knowledge. The tool provides intelligent viewing of operation dependencies, test results, coverage criteria, and failure analysis, enabling users to take further actions based on the insights gained. The IBM Consulting team has utilized this tool to test many IBM internal and client applications.

ACM Reference Format:

Diptikalyan Saha, Devika Sondhi, Swagatam Haldar, Margaret Mary Dhanaswamy, Aman Goyal, and Saritha Route. 2024. A Tool for Functional Testing of REST APIs. In *Proceedings of the 32nd ACM Symposium on the Foundations of Software Engineering (FSE '24)*, November 15–19, 2024, Porto de Galinhas, Brazil. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 INTRODUCTION

In order to ensure the reliability of REST APIs, extensive testing is crucial before their deployment, given their widespread usage and the multitude of applications that rely on them. To tackle this challenge, numerous automated test-generation techniques and tools have been developed. These approaches utilize the API specification as input and employ different strategies to generate test cases that thoroughly exercise the APIs. The primary objectives of these techniques are to enhance coverage and identify bugs. These techniques vary in the success they achieve with respect to these

goals [9]. However, it is important to note that these techniques often overlook the practical needs of functional testing for REST APIs as developers commonly perform.

This work concentrates on facilitating the automated generation of functional test suites specifically designed for REST APIs. The tool incorporates insights gained from interactions with IBM professionals. For instance, practitioners emphasized the significance of including tests in a functional test suite that not only covers individual API operations but also encompasses sequences of operations that encompass functional scenarios. These functional scenarios typically revolve around the create-read-update-delete (CRUD) operations performed on the API-managed resources such as accounts, products, and orders. Adopting a resource-centric perspective is critical for constructing effective functional test suites that can be readily employed for regression testing as APIs evolve.

We implemented the notion of automated functional testing in a tool called RAFT (REST API Functional Tester). RAFT takes as input the OpenAPI specification of the API under test (AUT) and automatically generates a test suite of functional test cases for the AUT; the generated tests cover individual API operations as well as sequences of operations in a systematic and well-defined manner.

This paper gives an overview of RAFT, describing its components and underlying principles, and highlights the usage flow and flexibility that our tool interface provides. We discuss how the user interaction with the tool interface provides a solution to the various challenges that testers face in a practical setup. These challenges pertain to the API specification not being written correctly, the need for customized data inputs, testing specific scenarios, and so on. The interface allows a human in the loop to analyze, apply filters, and edit the intermediate outcome from RAFT, right from passing the API specification to execution of test cases. The demo video of RAFT is available at [3].

2 RAFT ARCHITECTURE AND DETAILS

The architecture of RAFT is presented in Fig. 1. RAFT takes the API specification as the input, and the run-time information to execute the API. The outcome is specification analysis which can help in improving the specification, a dependency view that can help in understanding how operations interact within the API, a set of test cases, their execution results, coverage metrics, and failure analysis performed on these tests. To illustrate our approach, we use the Petstore API [13] as the running example. The API specification consists of 20 operations and six schemas. Fig. 2 lists the operations grouped by four resources (Pet, Order, Inventory, and User) managed by the API.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2024 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

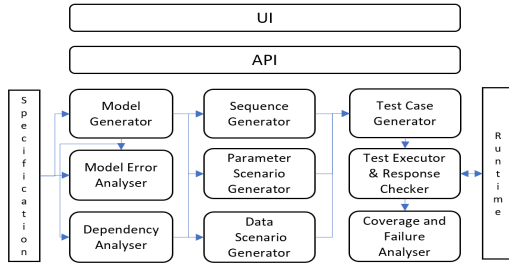


Figure 1: Architecture

Resource	Operations	Resource	Operations
/pet	findPetsByStatus deletePet getPetById uploadFile findPetsByTags addPet updatePet updatePetWithForm	/store/inventory	getInventory getUserByName getPetById updateUser createUser logoutUser createUser createUsersWithArrayInput createUsersWithListInput loginUser
/order	placeOrder getOrderByById deleteOrder		

Figure 2: Operations of the Petstore API grouped by resources

Operation-level	Resource	Sequence
addPet	order	addPet
addPet, deletePet		uploadFile
addPet, findPetsByStatus		updatePetWithForm
addPet, findPetsByTags		placeOrder
addPet, getPetById		getOrderByById
addPet, placeOrder		deleteOrder
addPet, placeOrder, deleteOrder	pet	addPet
addPet, placeOrder, getOrderByById		uploadFile
addPet, updatePet		updatePetWithForm
addPet, updatePetWithForm		getPetById
addPet, uploadFile		updatePet
createUser, deleteUser		deletePet
createUser, getUserByName	user	createUser
createUser, updateUser		getUserByName
createUsersWithArrayInput		updateUser
createUsersWithListInput		deleteUser
getInventory		

Figure 3: (A) Operation-level sequences and (B) resource-based sequences.

Specification Analysis. RAFT processes the specification to create a model that captures all operations and their parameter details such as types, mandatory/optional attributes, and constraints at various levels (individual parameters and multiple parameters), along with all schemas. The constraints are either pre-populated from the specification file or left to the user to populate. Uniqueness constraints and inter-parameter dependencies are examples of such user-defined constraints.

The model error analysis component performs several analyses to determine missing output parameters, missing required attributes, missing error codes, schema without properties, missing referenced schema, missing operation ids, and most importantly missing parameter constraints. They are also incrementally checked

with every user modification to the model, helping users to improve the model/specification.

Dependency Analysis. To construct operation sequences for a resource, RAFT first identifies resources. Recent related works such as MOREST [11] capture the relationship between operation and schema. Here we distinguish between the schema and resources. Schema refers to the syntactic entity used to define the input and output parameter types, whereas, resources is a logical entity. For example, the three schemas Category, APIResponse, Tag in Petstore are not resources.

Our algorithm identifies a resource by the maximal length operation path (removing versions and path parameters) that ends in a noun (resource names are typically nouns) or has multiple operations in it. The operations for the resource are all operations that are associated with the suffix of any path that has the resource as a prefix. See Fig. 2 for Petstore resources.

Once we obtain the group of operations associated with a resource, there are two additional challenges: how to order these operations and how to create pre-requisite sequences for an operation. We address these challenges based on a few types of dependencies, described below.

The **resource-based producer-consumer dependency (RPC)** occurs between an operation that creates the resource (producer) and the operation that accesses/uses the resource (consumer). The consumption is done through the *id* field of a resource, which uniquely identifies a resource. There are three instances of such a relationship in our sample test case - placeOrder (consumer) requires a Pet (producer) where placeOrder.body.petId (consumed parameter) refers to addPet.200.id (produced parameter). A similar relationship exists between placeOrder and getOrderByById and between placeOrder and deleteOrder where placeOrder acts as a producer for Order.

Data availability (DA) relationship. A consumer can also search for an object through other non-id fields. The success of these consumer operations at runtime depends on whether it finds the object. This is ensured by having an equality relationship between the producer's input or output to the consumer's input. An example of such a relationship exists between addPet.200.status and findPetsByStatus.query.status[].

RAFT solves these important challenges to infer the dependencies: (1) identifying producer operations, (2) determining 'id' fields, and (2) matching parameters.

Producers POST operations usually create resources. However, developers use POST operations not only to create resources but to update them and perform other functional operations. To obtain producers and create sequences, we need a fine-grained sub-typing of operations. There are three notions of operation sub-typing:

- **Cardinality:** Whether the operation works on a single resource or multiple resources. For example, createPet creates one resource whereas createUsersWithArrayInput creates multiple resources. This can be easily inferred by the presence of schema arrays in either input or output. We use *one* and *all* suffixes with operation types to denote these categories of subtypes.
- **CRUD-Anomaly:** Let us consider the POST operations uploadFile and updatePetWithForm that update the Pet resource. Beyond Petstore, we have seen a PUT operation semantically acts as a GET

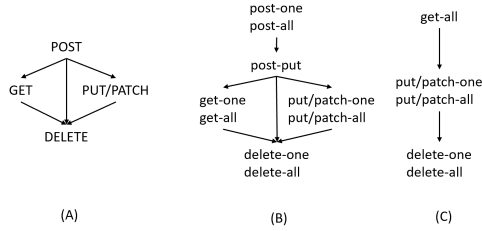


Figure 4: (A) CRUD partial order, (B) subtype partial order, and (C) alternate order in the absence of POST operations.

operation because the developer wanted to send an object in the request body. Such anomaly happens in practice and our algorithm uses input-output pattern-based matching and keyword-based matching to find such anomalies. E.g., `updatePetWithForm` takes `petId` as a parameter and `uploadFile` does not return an `id`. Note that, the above operations are subtyped as ‘post-put’ and they are not considered for producers as they are not creators.

- **Functional Operation:** Consider the GET operations `loginUser` and `logoutUser`. The verb in these operations does not match with any synonym of retrieval operation and therefore they are considered functional operations which is more relevant to application functionality than the CRUD functionality.

The producer is typically defined as operations with *post-one* or *post-all* subtype. In RAFT, users typically change the default configuration to include *get-all* if no POST operation exists for a resource. *Id-fields/Id-Param*. We use a preference order (1) type *UUID*, (2) names containing substring such as *id*, *code*, *name*, *timestamp*, and (3) in their absence, the most frequently accessed name matching field from other operations to select the best id field. This simple heuristics works surprisingly well in practice.

Parameter matching. We use the following features to match the consumer and producer parameters: (1) parameter name, (2) parameter type, (3) nouns in schema name (if any), (4) nouns in the resource name, and (5) resource. Consider the consumer field `placeOrder.body.petId` with features `{petId, integer, Order, Order, 'Order'}` and its corresponding matched producer field is `addPet.200.id` with features `{id, integer, Pet, Pet, /Pet}`. Given two fields with their features, RAFT uses a matching function that returns a rank. The rank is a weighted addition of the following matches: (1) type equality match (weight W_1) (2) field names match (exact or substring or acronym) (W_2) (3) schema or resource names match (W_3), (4) exact and acronym match between nouns in consumer field name to schema name/resource name match (W_4) (5) producer and consumer resource match if the consumer operation is a non-post operation (W_5). We give the following weights: $W_1=500$, $W_2=50$, $W_3=20$, $W_4=W_5=10$, prioritizing type match, param name match, and resource-name match in that order. The intuition for (4) is that consumers will typically use the schema/resource name in the field name to unambiguously identify the correct resource. Without this match, we do not consider the producer-consumer relationship across resources.

Resource-based Operation Sequencing. To perform functional testing, RAFT generates two types of sequences. It first generates test cases to test each operation (akin to unit testing). These sequences

contain the required prerequisites (using producer-consumer dependencies) followed by the target test operation, e.g., `addPet` is a prerequisite for target `getPetById`. Such sequences for petstore are shown in Figure 3(A).

RAFT generates functional sequences for each resource to check the feasibility of the functionality. Formally, a sequence of operations is said to be *complete* w.r.t a resource if it contains all types (i.e. HTTP methods) of operations (i.e. at least one operation of each type) related to the resource and *sound* if such operations maintain the CRUD partial order (see Fig. 4). A resource is said to be *functionally covered* if there exists at least one sequence (called a functional sequence) in the test suite which is sound and complete w.r.t the resource. The functional sequences for each resource are shown in Figure 3(B). Note they also contain the prerequisite operations as seen for resource order.

Given a resource and all operations related to it, one operation graph is created by adding edges between the operations for each subtype-partial order relationship (Figure 4(B) and (C)). This step will add `placeOrder` \rightarrow `getOrderById`, `placeOrder` \rightarrow `deleteOrder`, and `getOrderById` \rightarrow `deleteOrder` for the *order* resource. Next, RAFT introduces the producer nodes (if it does not exist already) for consumers in the operation graph from both *DA* and *RPC* relations, and recursively adds the producers of the newly added nodes along with their edges. `addPet` \rightarrow `placeOrder` is added using *RPC*. For any *post-one* operation added as a prerequisite, RAFT also adds (only for resource-based sequencing and not for operation-based) the *post-put* operations (and recursively their dependencies) for the same resource. This is based on our assumption that *post-put* operations are meant for required initialization (in contrast to PUT operation which is not essential for resource creation and setup). `addPet` \rightarrow `uploadFile`, `addPet` \rightarrow `updatePetWithForm` is added through this.

Next, three types of ordering edges are added to the graph. For multiple operations having the same subtype for a resource, we use a novel strategy for ordering the verbs of each operation based on their natural order. RAFT uses a transformer language model to find the permutation of the verbs with the highest probability. This will order `uploadFile` before `updateWithForm`. Even though login and logout are not part of execution in Petstore, this strategy will correctly identify the order `(login, logout)`.

To break the partial order between GET, PUT, and PATCH operations for resource, we add the ordering edges between operations in the type order `(GET, PUT, PATCH)`. For any operation edges between *post-one* and *post-put* operations added in the prerequisite, to keep them subsequent we also add outgoing edges from *post-put* operation to the same target operations of *post-one*. RAFT generates the final sequence based on the topological order of operations (see Fig. 3(B)).

Parameter Scenario. For each operation, RAFT creates three types of parameter scenarios: all mandatory parameters, all parameters, and each optional parameter along with required mandatory parameters.

Data Scenario. For each parameter, the data is automatically generated by respecting all constraints and types specified in the specification. There are multiple modes of data generation - 1) using

a constraint solver to create data satisfying model constraints. However, the generated data may not be realistic. To generate realistic data, we use 2) a repository-based strategy that retrieves realistic data from a large curated repository satisfying model constraints; 3) an LLM-based data generation strategy that generates realistic data satisfying model constraints encoded in prompts; 4) a mode where the data generation retrieves existing application data using the get-calls and mixes the data along with the generated data from above three strategies. In addition, the tool offers flexibility to create negative data scenarios.

Test Case Generation. Operation-level test cases are generated by considering only mandatory parameter scenarios for prerequisites and considering all possible parameter scenarios for the target operations. For each parameter scenario, only one data scenario is considered by default. When it comes to the functional scenario, only the mandatory parameters are taken into account for all operations. RAFT utilizes two default checks: the valid response code check, which verifies whether the response code is less than 400, and the schema conforming check, which ensures that the response structure conforms to the schema defined in the specification/model. To cater to the application's needs, the user has the flexibility to define new checks and select the operation context in which it will be applicable.

Test Execution and Response Checks. The test-executor component performs any required authentication and executes each operation, performs any checks, and then keeps track of the response data in a state such that the consumer parameter or checks can access the value from the state.

Coverage Metrics. The coverage metrics are divided into two types: test suite and runtime metrics. The test suite metrics encompass path, operation, and parameter coverage based on the test suite. On the other hand, the runtime metrics measure similar criteria regarding the percentage of successful runs, such as the percentage of operations covered successfully without any check failures. Additionally, it calculates response code coverage criteria.

3 INDUSTRY USAGE

RAFT is a very mature tool and has been successfully used by the IBM Consulting and Research teams to test 25 IBM internal APIs and more than 50 client APIs. The details of those results are confidential. However, they show the maturity and effectiveness of the tool. An IBM consulting testing team of 6 people uses it regularly typically along with one or two people from each client. The technical report in [3] contains more details.

4 KEY EXPERIMENTAL FINDINGS

Other than regular IBM internal usage, we experimented with 13 public APIs, that have been used in past research ([1, 4]) and compared against three state-of-the-art API testing tools, MOREST [11], EvoMaster (v1.6.0) [5], and RestTestGen [8, 16].

Metrics. Functional coverage (FC), is the percentage of resources functionally covered. The compactness of functional sequences (CFC) computes the ratio of the span of operations for a covered resource and the number of unique operations in the sequence related

Table 1: Average metric values across all benchmarks for all and non-trivial sequences (having at least two operations for resource)

Sequences	RAFT			MOREST			EvoMaster			RestTestGen		
	FC	CFC	SFC	FC	CFC	SFC	FC	CFC	SFC	FC	CFC	SFC
All	86	1.0	33	67	1.02	43	44	1.0	30	45	1.0	17
Non-trivial	81	1.01	31	50	1.6	21	15	1.0	50	12	1.0	0

to the resource averaged over all functional sequences. SFC measures the percentage of successful (all 2xx) functional sequences.

As seen in Table 1 RAFT is considerably better and more consistent than MOREST, EvoMaster, and RestTestGen in generating compact functional test sequences. RAFT also achieves greater parameter (100%), operation (100%), and successful operation coverage (65%) compared to MOREST (58%, 91%, 60%, respectively), and other tools. RAFT also achieves 86% accuracy in computing producer-consumer relationships. In terms of the percentage of sequences where the correct producer-consumer relationship is maintained, RAFT achieves (79%) compared to MOREST (17%), EvoMaster (20%), and RestTestGen (30%). Detail results are available at [3].

5 RELATED WORK

There are several commercial API testing tools [2] including Postman, RestAssured, Apigee, JMeter, SOAPUI, and API Connect. While these tools are useful for manually defining test cases, maintaining them, and viewing the test results, none of them support automated test case generation which can go across multiple operations. RAFT can export test cases in RestAssured and Gherkin format, providing interoperability.

Recently industry and academia have produced many test case generation based on black-box techniques. RESTler [6], MOREST [11], Evomaster [5], RestTestGen [16], bBOXRT [10], uTest [14], RESTest [12], Restats [7], HsuanFuzz [15], RestCT [17] are some of them. All the above test case generation strategies are not related to resource-based functional test case generation. Moreover, our tool produces accurate producer-consumer dependency compared to Morest, EvoMaster, and Restestgen. The details of the experiments are beyond the scope of this paper. Most of the above tools do not have any UI and are hard for industry practitioners to use. RestTest [12] tool is perhaps the most closely related to RAFT. However, their sequence generation is confined to each operation with the required pre-requisite and therefore similar to our operation-level test generation. Other than the algorithmic difference of functional sequence generation with precise producer-consumer dependency, our tool offers much better flexibility to modify the relevant dependency and parameter, data scenarios, and checks to modify the test suite.

6 CONCLUSION

In this paper, we presented a tool for automated REST API testing. The main functionality of the tool is the automatic generation of functional test cases from practitioners' perspectives. It uses novel algorithms for functional sequence generation and accurate parameter dependency inference. Our tool can handle multiple API specifications combining them into a single model and has been integrated into the Jenkins-based CI/CD pipeline for some clients. RAFT is proprietary and hence cannot be shared at this point.

REFERENCES

- [1] RestGo repository. https://github.com/codingsoo/REST_Go, 2022. [Online; accessed 19-August-2022].
- [2] 10 Best API Testing Tools In 2023 (SOAP And REST API Testing Tools). <https://www.softwaretestinghelp.com/api-testing-tools/>, 2023. [Online; accessed 26-May-2023].
- [3] RAFT materials. <https://github.com/diptikalyan/RAFT>, 2024. [Online; accessed 29-Jan-2024].
- [4] Andrea Arcuri. Evomaster: Evolutionary multi-context automated system test generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 394–397. IEEE, 2018.
- [5] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. Evomaster: A search-based system test generation tool. *Journal of Open Source Software*, 6(57):2153, 2021.
- [6] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *ICSE*, pages 748–758. IEEE, 2019.
- [7] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. Restats: A test coverage tool for restful apis. In *ICSME*, pages 594–598. IEEE, 2021.
- [8] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. Automated black-box testing of nominal and error scenarios in restful apis. *Software Testing, Verification and Reliability*, page e1808, 2022.
- [9] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated Test Generation for REST APIs: No Time to Rest Yet. In *ISSTA*, page 289–301, 2022.
- [10] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. A black box tool for robustness testing of rest services. *IEEE Access*, 9:24738–24754, 2021.
- [11] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. Morest: Model-based restful api testing with execution feedback. *arXiv:2204.12148*, 2022.
- [12] Alberto Martín-López, Sergio Segura, and Antonio Ruiz-Cortés. Resttest: automated black-box testing of restful web apis. In *ISSTA*, pages 682–685, 2021.
- [13] Petstore. PetStore Swagger Specification. <https://petstore.swagger.io/>, 2022. [Online; accessed 19-August-2022].
- [14] Stefano Russo. Assessing black-box test case generation techniques for microservices. In *QUATIC 2022*, page 46. Springer Nature, 2022.
- [15] Chung-Hsuan Tsai, Shi-Chun Tsai, and Shih-Kun Huang. Rest api fuzzing by coverage level guided blackbox testing. *arXiv preprint arXiv:2112.15485*, 2021.
- [16] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. Resttestgen: Automated black-box testing of restful apis. In *ICST*, pages 142–152. IEEE, 2020.
- [17] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. Combinatorial testing of restful apis. In *ICSE*, 2022.

7 APPENDIX

The user interaction flow for RAFT is shown in Figure 5.

The home page (see Fig. 6) of RAFT opens with two tabs - Applications and New. In the New tab, the user can register a new

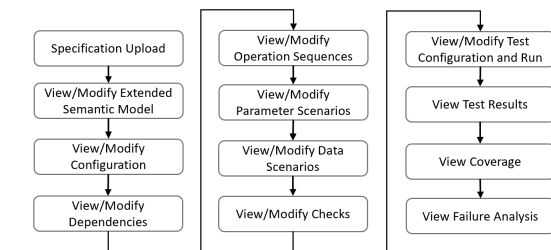


Figure 5: User flow

Figure 6: Upload API Specification

API by uploading its specification file. Currently, we support only OpenAPI/Swagger specifications (2.x and 3.x). The tool registers the API if a valid specification file is uploaded. RAFT also accepts zip of multiple API specifications which are parsed together to obtain a single model resolving external dependencies. All registered APIs appear as tiles in the application tab from which users can start testing any registered API (see Figure 7).

The user can now view the model (see Figure 8) derived from the specification and the corresponding error analysis results. The model is shown in a collapsable JSON editor which helps in finding and updating information in the model. Subsequently, the user can save the changes which also triggers the update of the error analysis.

Next, the user can view and edit the configuration which is pre-populated with some values which can help in identifying producer operations and help in sequencing the CRUD operations. Figure 9 shows a snapshot of the configuration which shows the configuration for subtype pattern for creating functional sequence. This configuration also helps in accepting authentication details, common header information, and any login credentials. The user can also specify three data generation modes - repository-based (default), LLM-based, and specification-based along with an option to use application data.

The semantic model and configuration are the inputs for generating the producer-consumer dependencies which can be viewed in the Dependencies tab (See Fig. 10). A d3-based graphical view shows the various operations and dependencies between operation parameters. The user can add/delete dependencies on the same page. The user may also see the change in the dependency computation in response to the change in the model and configuration. For example, the user may see that a producer is an input parameter of a POST operation because the specification does not have any response structure mentioned. The user can add the correct response structure in the model and re-infer the dependencies (see Figure 11).

In the next four tabs, the user can see the essential constituents of the test cases - sequences, parameters, data, and checks.

In the sequences tab, the user can view two types of sequences that are generated automatically - operation test sequences and resource-based functional test sequences with dependencies of parameters between such sequences in a graphical view (See Figure 12 and Figure 13, respectively). The parameter dependency view shows how parameter values will flow in an execution sequence. Even though the RAFT generates functional sequences by sequencing

Figure 7: Applications Tab

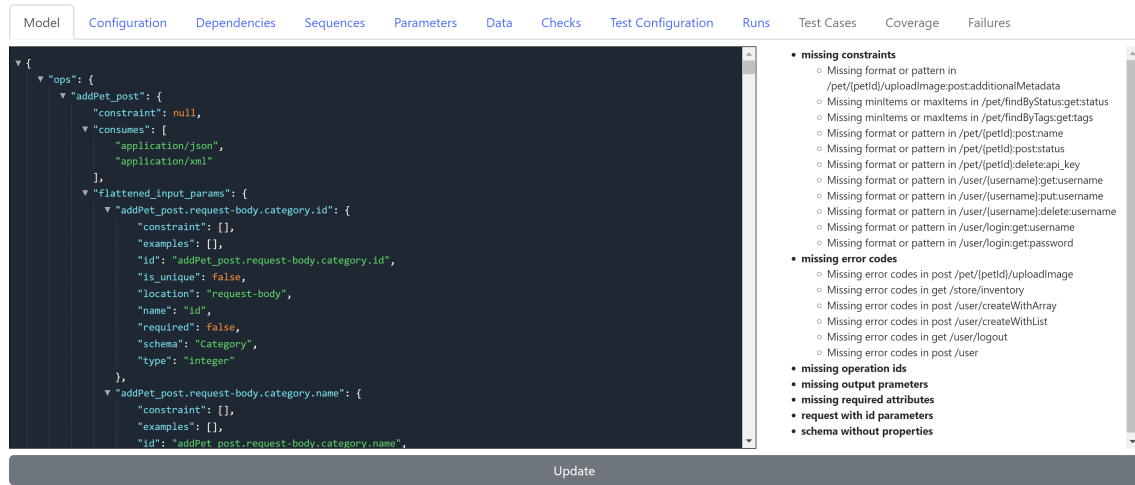


Figure 8: Model View

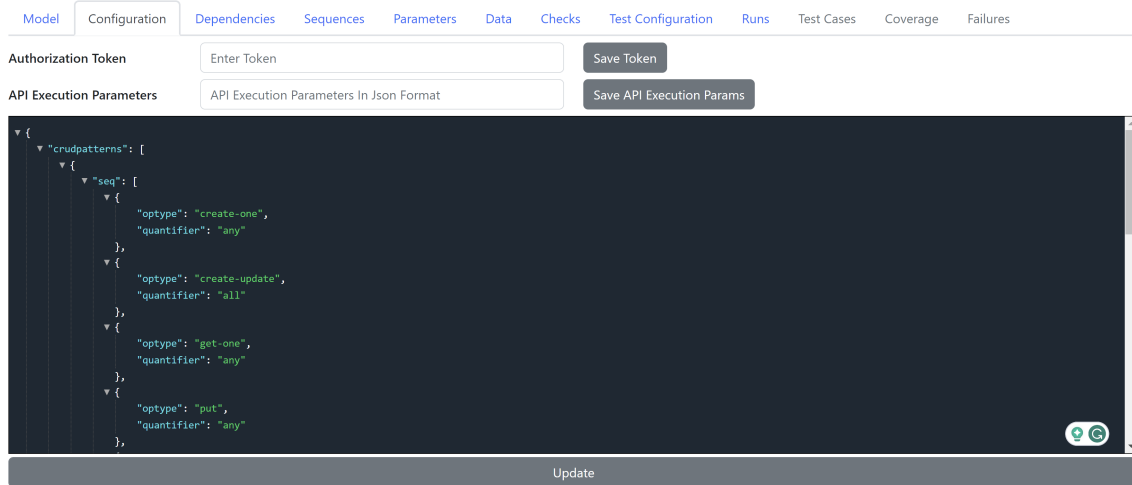


Figure 9: Configurations

per-resource CRUD operations, the user may have more application knowledge to create other important sequences which can be added through the UI. See Figure 14 for an example of an inventory-related sequence that the user has added with the add-sequence button.

The user can view the parameter scenario in the next tab which shows different parameter scenarios such as MANDATORY, OPTIONAL, and ALL parameters for every operation (see Figure 15). The user can re-compute the parameter scenario by changing the required attribute for each parameter in the model (using re-infer button) and create custom parameter scenarios (using '+' button). There are 4 contexts associated with parameters depicting how they can be used with the operations. Operations are classified into 3 types (Pre-requisite, Target, Functional) depending on the role they play in the sequences. Operation Sequences are used for unit testing and all types of parameter scenarios are associated with its

target operation (last operation). Pre-requisite operations in the operation sequences get only mandatory parameter scenarios by default. All operations in the functional sequences are called functional and only mandatory parameter scenario is associated with it, by default. However, the user can select one or more contexts for the parameter scenario using the context multi-select drop-down on the left.

RAFT pre-populates the data tab with the artificially created data as per specification for each operation. Figure 16 shows the output of the repository-based data generation. The color coding and tooltip signify how the data is generated. For example yellow represents retrieved data whereas pink refers to the categorical constraints or examples specified in the specification/model. The user can add/modify/delete/download data. This functionality helps the user to import pre-existing test data or try out new variations



Figure 10: Producer-Consumer Dependencies

Re-Infer Dependencies from Model

Save All Changes

Check to Delete

Producer	Consumer
<input type="checkbox"/> addPet_post/200.id	deletePet_delete/path.petId
<input type="checkbox"/> addPet_post/200.id	getPetById_get/path.petId
<input type="checkbox"/> addPet_post/200.id	placeOrder_post/request-body.petId
<input type="checkbox"/> addPet_post/200.id	updatePetWithForm_post/path.petId
<input type="checkbox"/> addPet_post/200.id	updatePet_put/request-body.id
<input type="checkbox"/> addPet_post/200.id	uploadFile_post/path.petId
<input type="checkbox"/> addPet_post/200.tags.array_item.name	findPetsByTags_get/query.tags.array_item
<input type="checkbox"/> createUser_post/request-body.id	updateUser_put/request-body.id

Producer

addPet_post

addPet_post/request-body.category.id

Consumer

addPet_post

addPet_post/request-body.category.id

Add Producer Consumer

Figure 11: Update Dependencies

that the tool does not generate automatically. The user can change the data using another algorithm using the configuration shown in Figure 17. The output of LLM-based data generation is shown in Figure 18. You can notice that LLM-based (we have used LLaMAv2 in WatsonX) generation produces much more realistic values than the repository-based algorithm. The user can automatically generate negative data scenarios. In Figure 19, you can see that the id-column has now various types of values. RAFT incorporates

20 variations of negative scenarios for numeric, categorical, string, and date data types.

By default, the tool shows two checks as described before (See Figure 20). The user has the ability to add binary expressions related to output parameters, response code, response time, response json, and concrete values. The tool can evaluate these expressions on responses and the state managed by the test executor. The user can also select the context in which the check is applicable. For example, the user can define a check involving response time and select "All"

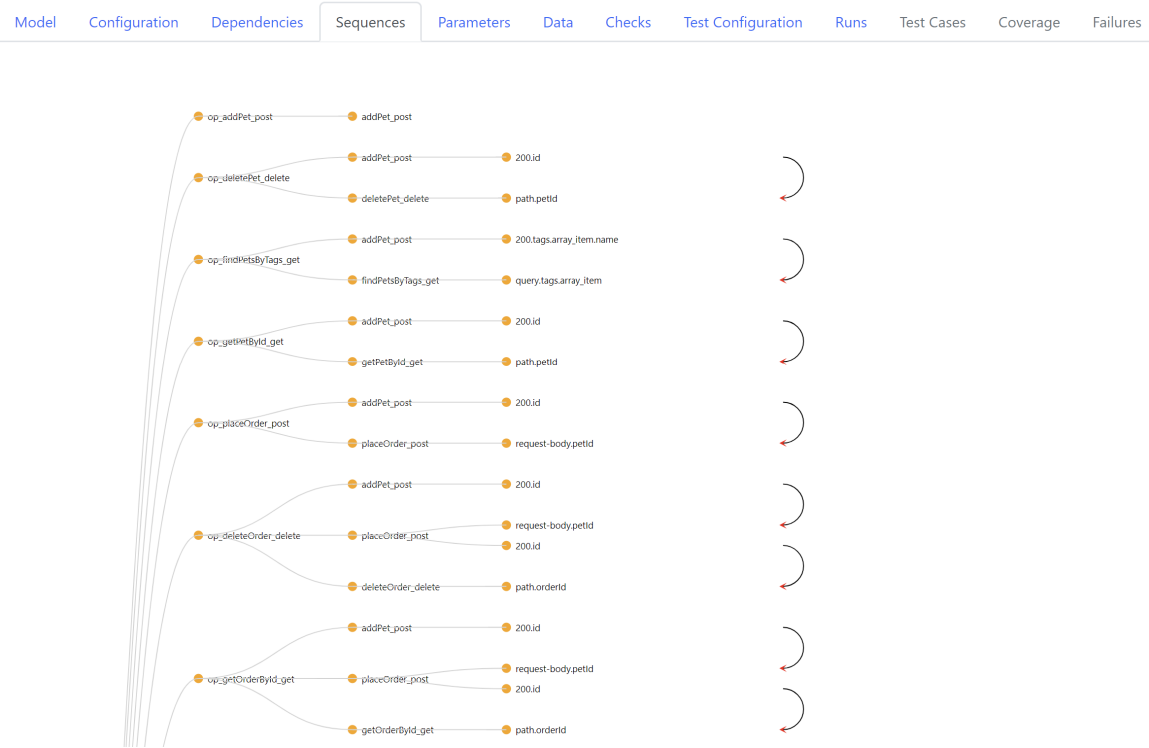


Figure 12: Operation Sequences

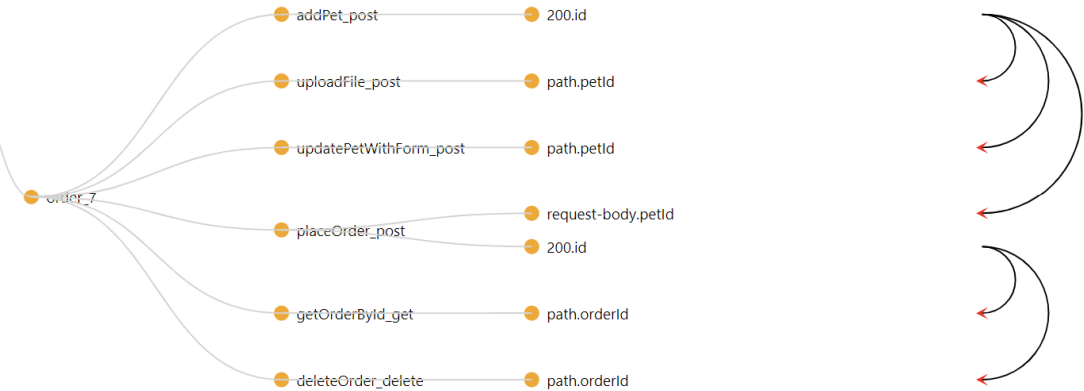


Figure 13: Functional Sequences

from the context list, which would make the check evaluated after every operation.

The test configuration page (see Figure 21) allows the tester to select sequences, parameter scenarios, data scenarios, and checks, and create test cases based on these selections. The user has two options: either to generate only the test cases (offline) or to generate and execute the test cases (online). The first option allows the user to generate the test cases offline even without the presence of runtime.

It's important to note that the user is not required to view all the previous pages and can directly access the test configuration page, schedule a run, check the results, and then potentially go back to modify the scenarios.

On the runs tab (see Figure 22), the user can view the most recent scheduled run, as well as all the historical runs for this API. The user can also execute or re-execute a run. This enables the user

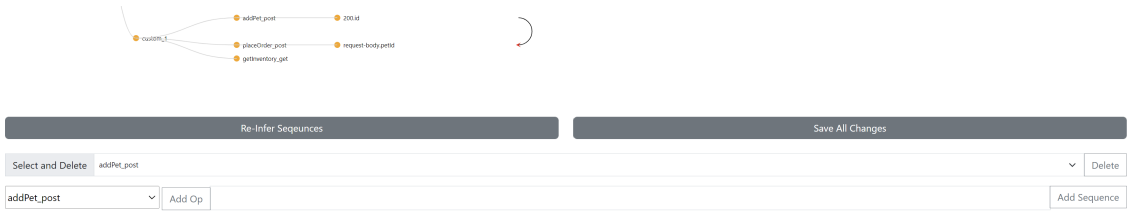


Figure 14: Custom Sequences

Figure 15 displays the 'Parameters' tab for the 'addPet_post' scenario. The table lists various parameters and their corresponding actions. The parameters include 'addPet_post.request-body.name', 'addPet_post.request-body.photoUrls.array.item', 'addPet_post.request-body.category.id', 'addPet_post.request-body.category.name', 'addPet_post.request-body.id', 'addPet_post.request-body.name', 'addPet_post.request-body.photoUrls.array.item.addPet_post.request-body.status', 'addPet_post.request-body.category.id.addPet_post.request-body.id.addPet_post.request-body.name.addPet_post.request-body.photoUrls.array.item.addPet_post.request-body.status', 'addPet_post.request-body.category.name.addPet_post.request-body.id.addPet_post.request-body.name.addPet_post.request-body.photoUrls.array.item.addPet_post.request-body.status', 'addPet_post.request-body.id.addPet_post.request-body.name.addPet_post.request-body.photoUrls.array.item.addPet_post.request-body.status.addPet_post.request-body.tags.array.item.id', and 'addPet_post.request-body.id.addPet_post.request-body.name.addPet_post.request-body.photoUrls.array.item.addPet_post.request-body.status.addPet_post.request-body.tags.array.item.name'. The actions are represented by icons for adding and deleting parameters.

Figure 15: Parameter Scenario

Figure 16 shows the 'Data' tab for the 'addPet_post' scenario. The table displays data generated from a repository, including status, id, name, and photoUrls. The data is organized into rows, each representing a different status and its associated data. The actions column provides options for adding and deleting data.

Figure 16: Repository-based Data Generation

Figure 17 displays the 'Configuration' tab for the tool. It includes fields for 'Authorization Token' and 'API Execution Parameters'. Below these fields, a JSON configuration is shown, detailing 'crudpatterns' and 'prodtypes'. The 'crudpatterns' field is set to '[3 items]', 'getallfordata' is set to 'false', 'datagenalgo' is set to '11m', and 'prodtypes' is set to ['create-one', 'create-all'].

Figure 17: Data Configuration







to execute previously generated test cases that haven't been run before or re-execute already executed test cases if there have been changes to the underlying API.

The user can view the results (see Figure 23) of a run and analyze them using filters at the operation, sequence, response-code level, or based on other check results. For each executed sequence, the user can view the request, response, checks, and check results corresponding to each operation. Alternatively, for a test case that hasn't been run, the user can view the request and checks.

The coverage tab displays the coverage results (see Figure 24) for different coverage metrics. Each coverage metric is accompanied by explanations that help the user understand their implications.

Model Configuration Dependencies Sequences Parameters Data Checks Test Configuration Runs Test Cases Coverage Failures

addPet_post

id	name	id	name	photoUrl[0]	status	id	name	Actions
1	Dogs	1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 
1	Dogs	2	Max the doggie	https://example.com/max.p	pending	2	Adventure dog	 
1	Dogs	3	Bella the doggie	https://example.com/bella.p	sold	3	Cute dog	 

+ Save All Generate Negative Data Generate Negative Data for All Ops Re-Generate Download CSV Download All CSV Choose File

Figure 18: LLM-based Data Generation

Model Configuration Dependencies Sequences Parameters Data Checks Test Configuration Runs Test Cases Coverage Failures

addPet_post



























id	name	id	name	photoUrl[0]	status	id	name	Actions
1	Dogs	1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 
1	Dogs	2	Max the doggie	https://example.com/max.p	pending	2	Adventure dog	 
1	Dogs	3	Bella the doggie	https://example.com/bella.p	sold	3	Cute dog	 
-7	Dogs	1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 
24414	Dogs	1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 
9.836798027735824e+24	Dogs	1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 
7x	Dogs	1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 
7	Dogs	1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 
8	Dogs	1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 
6	Dogs	1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 
1		1	Buddy the doggie	https://example.com/buddy	available	1	Puppy	 

Figure 19: Negative Data Generation

Model Configuration Dependencies Sequences Parameters Data Checks Test Configuration Runs Test Cases Coverage Failures

Description	Context	Operator	LHS	RHS	Actions
responseCodeValid	All Target Functional addPet_post	<	responsecode	400	 
schemaconformitycheck	All Target Functional addPet_post		schemaconformitycheck		 

+

Figure 20: Checks

The failures tab (See Figure 25) consolidates all unique failures in one place, providing the user with a clear view to take corrective action. It uses an error message-based clustering algorithm to create failure clusters.

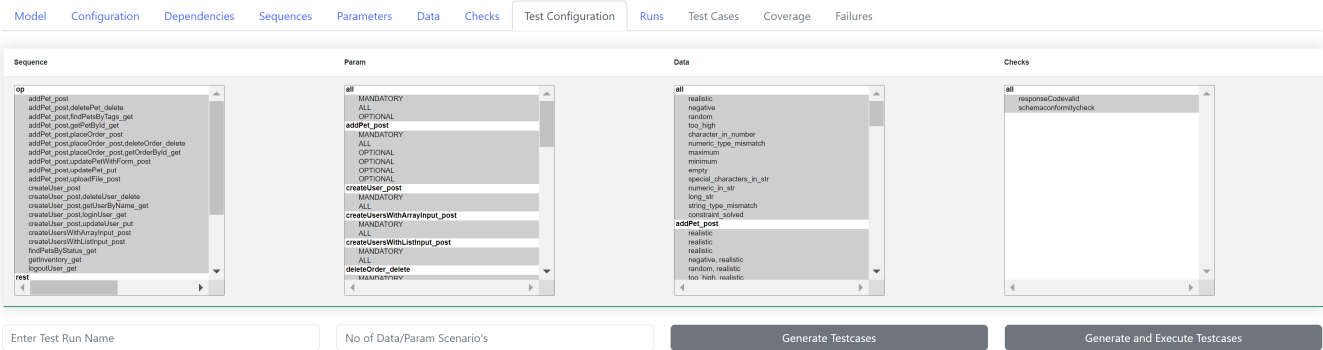


Figure 21: Test Configuration

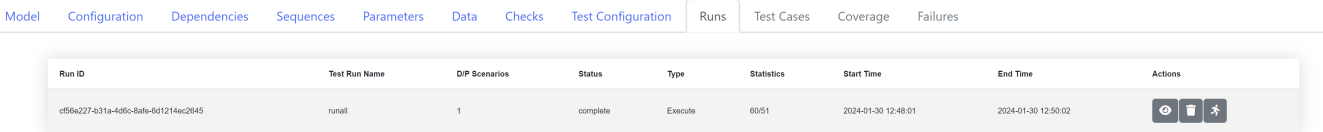


Figure 22: Run Summary

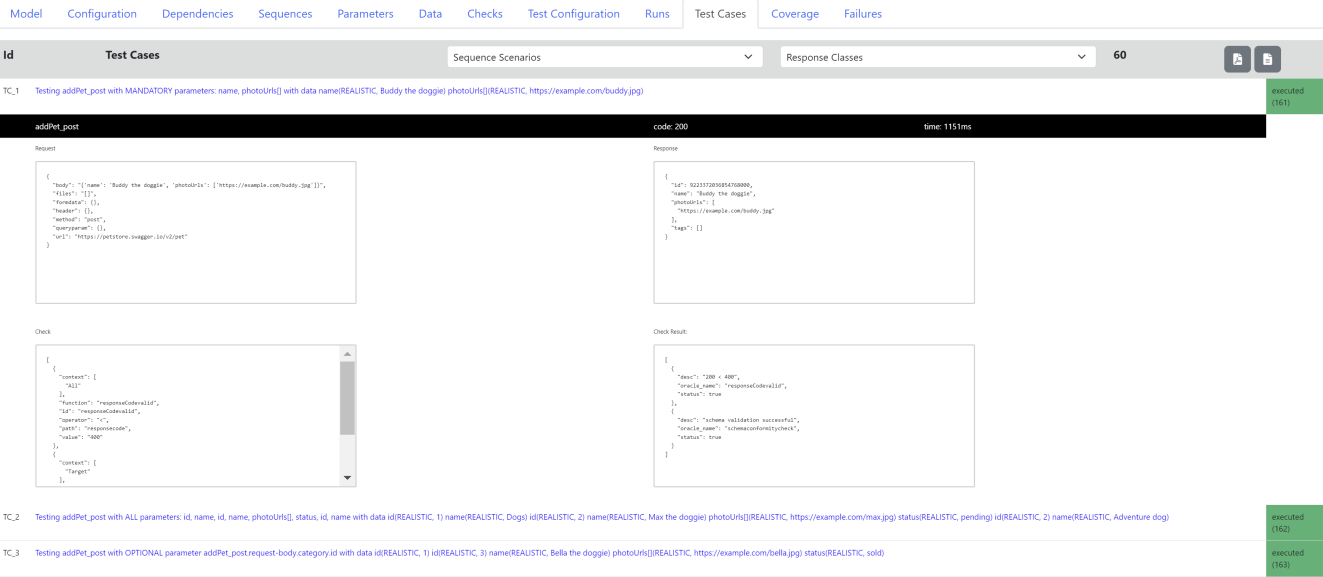


Figure 23: Test Cases

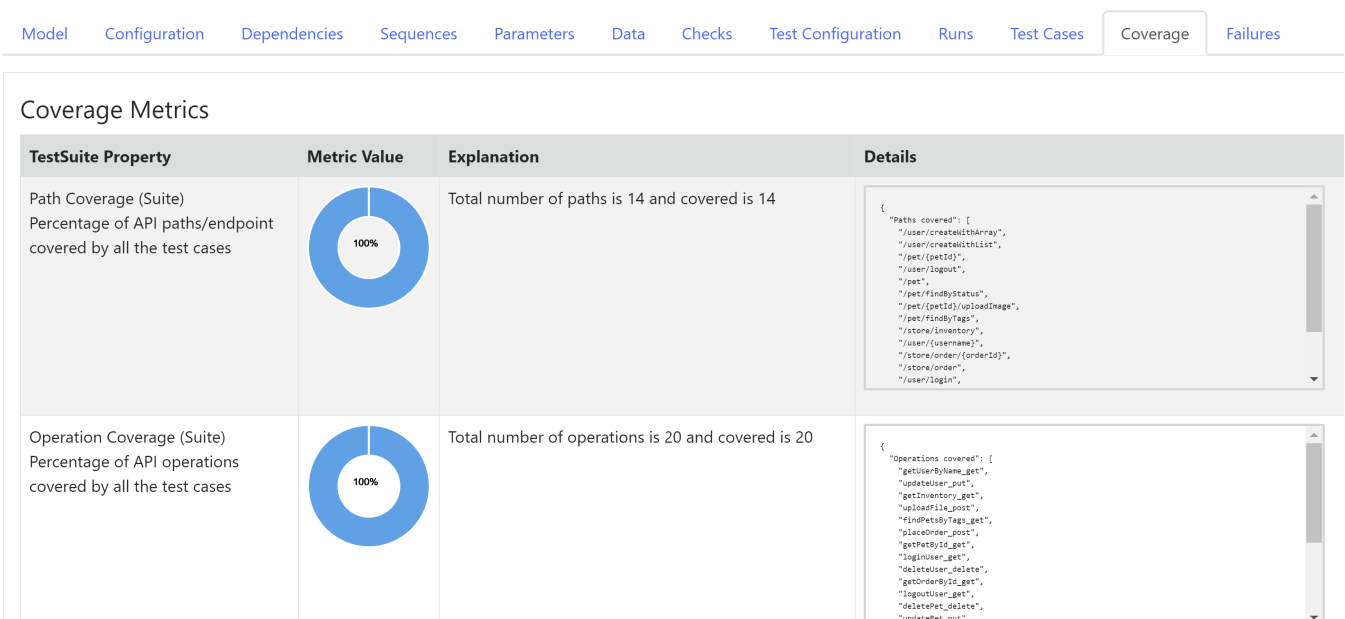


Figure 24: Coverage Metrics

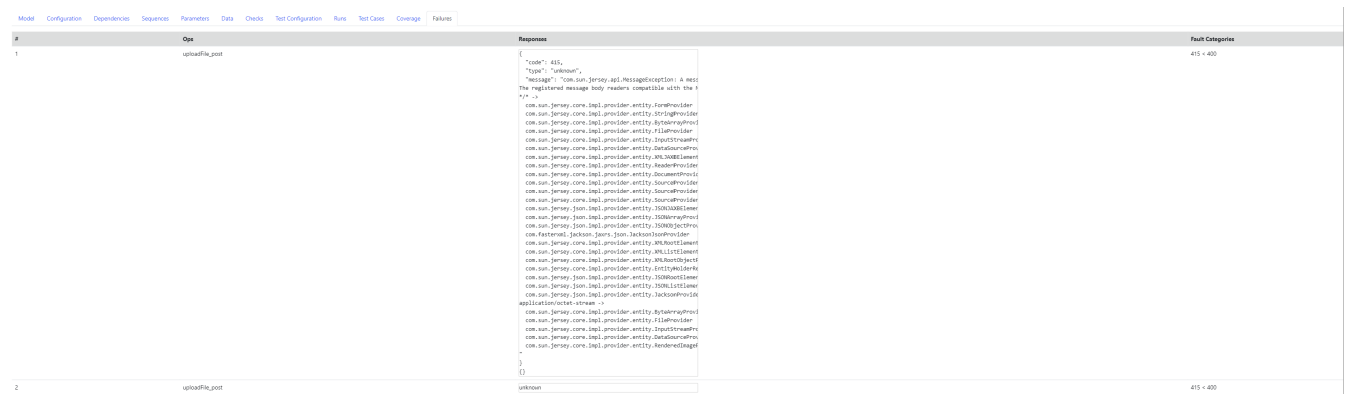


Figure 25: Failure Analysis