Satellite Configuration Service - Automation Testing Challenge

Project Overview

The project involves validating and testing a RESTful API for satellite configuration management. Key tasks include developing automated tests for API operations, identifying and documenting issues in the API documentation, and creating a test plan for a mock frontend release linked to the API.

Test Strategy

Objectives :

- Validate the API's functionality for managing satellite configurations.
- Verify frontend functionality, ensuring users can create and retrieve configurations seamlessly.
- Enforce data integrity, error handling, and API compliance.

Scope :

- API Testing
  - Functional Testing (CRUD operations).
  - Negative Testing (boundary conditions, invalid data).
  - Performance Testing (server under load for creating configurations).

- Frontend Testing
  - Validate input fields and form submission.
  - Verify UI responsiveness and behavior.

- Documentation Testing
  - Compare actual API responses against provided documentation.

Part 1 : API Testing

- API Automated Tests

A suite of automated tests in *Robot Framework* has been developed to validate API functionality and error handling. These tests cover standard endpoint operations (*GET*, *POST*, *PUT*, *DELETE*), boundary conditions, responses to invalid data formats, and deliberate error scenarios.

The *Robot Framework* tests can be run manually or configured to execute via *Jenkins*, which automatically retrieves the test code and related resources from the *GitHub* repository at https://github.com/aristotelis-metsinis/satellite-configuration-service

In the latter case, the *Jenkins* job can be configured to use the following *shell* script :

```
#!/bin/bash

# Set the path to the project directory within the Jenkins workspace
PROJECT_DIR="$WORKSPACE"

# Check if virtual environment exists; if not, create it
if [ ! -d "$PROJECT_DIR/.venv" ]; then
    python3 -m venv "$PROJECT_DIR/.venv"
fi

# Activate virtual environment
source "$PROJECT_DIR/.venv/bin/activate"

# Install dependencies if not already installed
if [ ! -f "$PROJECT_DIR/requirements.txt" ]; then
    echo "requirements.txt not found!"
else
    pip install -r "$PROJECT_DIR/requirements.txt"
fi

# Start the server in the background and capture the PID
"$PROJECT_DIR"/server/automation-tester-challenge-project --port 1234 > /dev/null 2>&1 &
# Capture the PID of the background process
SERVER_PID=$!

# Sleep for 2 seconds
sleep 2

# Change directory to the project folder within the workspace
cd "$PROJECT_DIR"

# Run Robot tests
.venv/bin/python3 -m robot --NoStatusRC -d log tests/
```

```
# Deactivate virtual environment
deactivate

# Forcefully kill the server
kill -9 $SERVER_PID

# Verify if the process has been killed
if ps -p $SERVER_PID > /dev/null; then
   echo "Server is still running"
else
   echo "Server has been successfully killed"
fi
```

- Additional Test Case Scenarios

Input Validation :	• Test creating, updating, or deleting configurations with excessively long *name* fields.
Extending Positive Test Scenario with Data Validation :	• The positive “Create Configuration” test scenario can be extended to validate the data of the newly created satellite configuration.
Boundary Conditions :	• Test the lowest and highest possible values for *cospar identifier* components (e.g., minimum “YYYY” year, valid three-digit “DDD”, etc.).
Concurrent Requests :	• Test simultaneous *POST*, *PUT*, or *DELETE* requests for the same resource. • Verify rate limiting by sending rapid bursts of requests to the endpoints.
Security Scenarios :	• Test for *SQL Injection* via fields like *cospar identifier* or *name* (e.g., submitting “1 OR 1=1”). • Test for *XSS* vulnerabilities by submitting malicious input in fields like *name*. • Test for *CSRF* vulnerabilities by attempting to perform actions (e.g., *DELETE*) without proper *CSRF* tokens. **Comment** : A limited set of (trivial) “security” automated tests has been developed to verify the server's response in cases of *SQL Injection* and *Cross-Site Scripting*. In these cases, the server responds with a successful message, although an error response is expected. In contrast, the server returns an error message in the “Authorization” test, as expected.
Error Injection :	• Test the behavior of the *GET*, *PUT*, and *DELETE* endpoints with invalid *id* values (e.g., non-numeric IDs, IDs with special characters). • Simulate API downtime and test client handling of server errors (500).

Part 2 : API Documentation Discrepancies, Observations, and Improvement Suggestions

This section highlights inconsistencies between the documented request/response formats and the actual API responses, including discrepancies in field formats, incorrect or missing fields, as well as suggestions for improving the API specification and documentation based on the observed server behavior and error-handling responses.

- confirmation messages

Endpoint	*POST /configs*
Issue	The response body includes JSON object with a confirmation message : “*Mission config created successfully”* while the documentation specifies : “*Mission configuration created successfully.”*
Suggested Correction	Update the documentation to specify the correct confirmation message or update the API to match the documented one.

Endpoint	*DELETE /configs/{id}*
Issue	The response body includes JSON object with a confirmation message : “*Mission config deleted successfully”* while the documentation specifies : “*Mission configuration deleted successfully.”*
Suggested Correction	Update the documentation to specify the correct confirmation message or update the API to match the documented one.

Endpoint	*PUT /configs/{id}*
Issue	The response body includes JSON object with a confirmation message : “*Mission config updated successfully”* while the documentation specifies : “*Mission configuration updated successfully.”*
Suggested Correction	Update the documentation to specify the correct confirmation message or update the API to match the documented one.

- maximum configurations

Endpoint	*POST /configs*
Issue	The service can store a maximum of “*6*” configurations at a time. If this limit is exceeded then an error is returned, while the documentation specifies that the service can store a maximum of “*10*” configurations at a time.
Suggested Correction	Update the documentation to specify the correct maximum number of configurations at a time or update the API to match the documented one.

- *payload* type field values

Endpoint	*POST /configs* and *PUT /configs/{id}*
Issue	The service can support the following types (values) of the *payload* JSON field in the request body : *"OPTICAL", "SAR", "TELECOM"* while the documentation specifies : *"OPTIC", "SARS", "TELECOM"*
Suggested Correction	Update the documentation to specify the correct types (values) of the *payload* JSON field or update the API to match the documented ones.
Comment	The tests primarily use the “actual” *payload* types, *OPTICAL* and *SAR*, in the request body rather than the “expected” (documented) types (*OPTIC, SARS*), to facilitate the development of “functional” tests for the purposes of presenting this demo project. While making assumptions about system design is generally **not** ideal and should be **avoided** in practice, one test compares both the “actual” and “expected” *payload* type values, highlighting discrepancies between the specifications and the developed API.

- *payload* field name

Endpoint	*POST /configs* and *PUT /configs/{id}*
Issue	The service can support the following name of the *payload* JSON field in the request body : “*type*” while the documentation specifies : *"payload_type"*
Suggested Correction	Update the documentation to specify the correct name of the *payload* JSON field or update the API to match the documented one.
Comment	The tests use the “actual” *payload* field name, *type*, in the request body instead of the

	“expected” (documented) field name, *payload_type*, to facilitate the development of “functional” tests for the purposes of presenting this demo project. While making assumptions about system design is generally **not** ideal and should be **avoided**, it is worth noting that the *payload* field name in the response body of REST API operations retrieving configurations is currently *type*. Therefore, it is expected that the correct naming convention for this JSON field will be clarified with the business owner and developer during the specification review phase, **prior** to test case development.

- *cospar identifier* pattern

Endpoint	*POST /configs* and *PUT /configs/{id}*
Issue	The service can support the following value pattern for the *cospar identifier* JSON field in the request body : “*YYYY-###XX”* where “*YYYY*” is the launch year, “*###*” is a three-digit mission code, and “*XX*” is a two-letter identifier while the documentation specifies : “*YYYY-###XXX”*
Suggested Correction	Update the documentation to specify the correct value pattern for the *cospar identifier* JSON field or update the API to match the documented one.
Comment	The tests primarily use the “actual” *cospar identifier* pattern, *YYYY-###XX*, rather than the “expected” (documented) pattern (*YYYY-###XXX*), to facilitate the development of “functional” tests for the purposes of presenting this demo project. While making assumptions about system design is generally **not** ideal and should be **avoided** in practice, tests validate both the “actual” and “expected” *cospar identifier* patterns, highlighting discrepancies between the specifications and the developed API.

- *cospar identifier* uniqueness

Endpoint	*POST /configs* and *PUT /configs/{id}*
Issue	The API documentation does not specify whether *cospar identifier* uniqueness is enforced.
Suggested Correction	Update the documentation to clarify this point.

- mission *name* length and allowed characters

Endpoint	*POST /configs* and *PUT /configs/{id}*
Issue	The API documentation does not specify the maximum character length or the allowed characters for the *name* JSON field in the request body.

Suggested Correction	Update the documentation to clearly state the maximum length for the *name* JSON field.

- mission *id* assignment and reuse

Endpoint	*GET /configs/{id}* and *DELETE /configs/{id}* and *PUT /configs/{id}*
Issue	The business logic for "*id*" calculation is not detailed in the API documentation. For example, the "configuration" service automatically assigns a unique ID to each new configuration using the formula N' = N + 1, where N is the maximum ID of all existing configurations. Assigned IDs are permanent and are not recalculated after a configuration is deleted, which can result in non-sequential IDs with gaps. Additionally, the API documentation does not clarify whether an ID can be reused after deletion.
Suggested Correction	Update the documentation to clarify this point.

- case sensitive *payload* type and *cospar identifier* field values

Endpoint	*POST /configs* and *PUT /configs/{id}*
Issue	The API documentation does not clearly specify whether the values for the *payload* type and *cospar identifier* JSON fields in the request body are case-sensitive. Currently, these values are case-sensitive, and the server responds with an error message when the provided values do not adhere to the expected case.
Suggested Correction	Update the documentation to clearly state that *payload* type and *cospar identifier* JSON field values are case sensitive.

- *payload* type, *cospar identifier* and mission *name* required "Create" JSON fields

Endpoint	*POST /configs*
Issue	The API documentation does not clearly specify whether the *payload* type, *cospar identifier*, and *name* JSON fields in the request body are required. Currently, these fields are mandatory, and the server responds with an error if any of them are missing.
Suggested Correction	Update the documentation to cleary state that the *payload* type, *cospar identifier* and mission *name* are required JSON fields.

- *payload* type, *cospar identifier* and mission *name* optional "Update" JSON fields

Endpoint	*PUT /configs/{id}*
Issue	The API documentation specifies that the *payload* type, *cospar identifier*, and *name* JSON fields in the request body are optional. However, these fields are mandatory, and the server responds with an error if any of

	them are missing.
Suggested Correction	Update the documentation to clearly specify that the *payload* type, *cospar identifier* and mission *name* are required JSON fields, or modify the API to handle these fields as optional.
Comment	The tests generally update a configuration by submitting all fields, assuming that the required or optional JSON fields will be clarified with the business owner and developer during the specification review phase, **prior** to test case development. This approach allowed to develop “functional” tests for the purposes of presenting this demo project, although making assumptions about the system design is **not** ideal and should be **avoided** in practice. However, there are tests that update with partial data, which reveal discrepancies between the specifications and the developed API.

- *payload* type, *cospar identifier* and mission *name* empty or null “Create” JSON fields

Endpoint	*POST /configs*
Issue	The API documentation does not clearly specify the server behavior when the *payload* type, *cospar identifier*, and *name* JSON fields in the request body have either empty or null values. Currently, the server responds with an error message in all such cases, as follows : *invalid request due to <> is required*
Suggested Correction	Update the documentation to clearly state the server behavior under these conditions.

- *payload* type, *cospar identifier* and mission *name* empty or null “Update” JSON fields

Endpoint	*PUT /configs/{id}*
Issue	The API documentation does not clearly specify the server behavior when the *payload* type, *cospar identifier*, and *name* JSON fields in the request body have either empty or null values. Currently, the server responds with an error message in such cases, as shown below : • when field value is empty : *invalid request due to invalid <>* • when field value is null : *invalid request due to <> is required* apart from the case where the *name* JSON field value is empty, in which case the server successfully updates the requested *ID*, but the updated JSON object no longer contains the *name* field, as shown in the following sample : *{* *"id": 3,* *"type": "SAR",* *"cospar_id": "2025-005XY"* *}*
Suggested	Update the documentation to clearly state the server behavior under these conditions.

Correction	

- successful and error scenarios

Endpoint	*GET /configs* and *GET /configs/{id}* and *POST /configs* and *DELETE /configs/{id}* and *PUT /configs/{id}*
Issue	The API documentation provides an overview of the available endpoints, request and response formats, and example usage, with a primary **focus** on **successful** scenarios.
Suggested Correction	Update the documentation to include an overview of the available endpoints, request and response formats, and example usage, with a focus on **both** successful and **error** scenarios.

- error messages and codes

Endpoint	*GET /configs* and *GET /configs/{id}* and *POST /configs* and *DELETE /configs/{id}* and *PUT /configs/{id}*
Issue	As previously mentioned, the API documentation does not describe the response format or status codes for various error conditions (e.g., 400 for invalid payload, 404 for missing resources). It also lacks information on how extra fields or missing optional fields in the request body are handled, whether there is a standard structure for internal server errors (500), and if specific error codes (e.g., 422 for validation errors) are used.
Suggested Correction	Update the documentation to clarify these points and ensure clear differentiation between client-side (4xx) and server-side (5xx) errors.

- sample responses

Endpoint	*GET /configs* and *GET /configs/{id}* and *POST /configs* and *DELETE /configs/{id}* and *PUT /configs/{id}*
Issue	The API documentation does not clearly describe the format, the field names and values of the JSON response body under different scenarios. For example the server responds with a body like the following ones in succesful scenarios : *{* *"meta": null,* *"data": {* *"message": "Mission config updated successfully"* *},* *"errors": null* *}* or *{* *"meta": null,* *"data": {*

	"id": 1, *"name": "Satellite1",* *"type": "OPTICAL",* *"cospar_id": "2024-003ZZ"* *},* *"errors": null* *}* or *{* *"meta": null,* *"data": [* *{* *"id": 1,* *"name": "Satellite1",* *"type": "OPTICAL",* *"cospar_id": "2024-003ZZ"* *},* *{* *"id": 2,* *"name": "Satellite2",* *"type": "OPTICAL",* *"cospar_id": "2023-001BC"* *}* *],* *"errors": null* *}* while with a body like the following one in error scenarios : *{* *"meta": null,* *"data": null,* *"errors": [* *{* *"message": "invalid request due to name is required",* *"source": "data-server"* *}* *]* *}* The *meta*, *data*, *errors* JSON fields and their subfields such the *source* are not described in detail. The conditions under which the *data* or *errors* JSON fields appear as JSON arrays are not explained.
Suggested Correction	Update the documentation to clearly describe the format, field names, and values of the JSON response body, along with the response status code, under different scenarios.

- capitalization in response messages

Endpoint	*GET /configs* and *GET /configs/{id}* and *POST /configs* and *DELETE /configs/{id}* and *PUT /configs/{id}*
Issue	The response body includes a JSON object with a message that is inconsistent in terms of capitalization. For example : • *"Mission configuration deleted successfully."* • *"invalid request due to mission config database is full."*
Suggested Correction	Apply a consistent format for all response messages.

- endpoints versus *HTTP* methods

Endpoint	*GET /configs* and *GET /configs/{id}* and *POST /configs* and *DELETE /configs/{id}* and *PUT /configs/{id}*
Issue	The API documentation does not describe the server behavior when an *HTTP* method different from the specified one is used to request data from a specified resource, as shown in the following matrix :

endpoint	*GET*	*POST*	*DELETE*	*PUT*
/configs (without JSON request body)	List of configurations retrieved	Invalid request	Method not allowed	Method not allowed
/configs/{id} (without JSON request body)	Configuration retrieved	Method not allowed	Configuration deleted	Invalid request
/configs (with JSON request body)	List of configurations retrieved	Configuration created	Method not allowed	Method not allowed
/configs/{id} (with JSON request body)	Configuration retrieved	Method not allowed	Configuration deleted	Configuration updated

Suggested Correction	Update the documentation to describe the server behavior under these conditions.

- authentication and authorization

Endpoint	*GET /configs* and *GET /configs/{id}* and *POST /configs* and *DELETE /configs/{id}* and *PUT /configs/{id}*
Issue	The API specifications do not mention if authentication (e.g., tokens, API keys) or user roles are required.
Suggested	Update the documentation to include details on authentication mechanisms and

Correction	authorization rules (e.g., admin-only access to *POST*, *PUT*, *DELETE*).

- concurrency and rate limiting

Endpoint	*GET /configs* and *GET /configs/{id}* and *POST /configs* and *DELETE /configs/{id}* and *PUT /configs/{id}*
Issue	The API specifications do not describe how the API handles concurrent requests or race conditions (e.g., updating or deleting the same resource simultaneously) or whether the API has rate limits or throttling to prevent abuse.
Suggested Correction	Update the documentation to include details on these points.

- deprecation handling

Endpoint	*GET /configs* and *GET /configs/{id}* and *POST /configs* and *DELETE /configs/{id}* and *PUT /configs/{id}*
Issue	The API specifications do not describe how changes to the API (e.g., field renaming) will be communicated or handled.
Suggested Correction	Update the documentation to include details on these points.

- pagination

Endpoint	*GET /configs*
Issue	The API specifications do not clarify if pagination or a limit on the number of records returned is implemented for the *GET* endpoint that retrieves a list of all satellite configurations.
Suggested Correction	Update the documentation to include details on these points.

Part 3: Test Plan for Mock Frontend Release

This section outlines a test plan for the frontend, focusing on functional, usability, and edge-case testing. Key objectives include verifying the creation and retrieval of configurations, assessing the usability of *Create* and *Search* functionalities, handling invalid operation scenarios, and proposing UI/UX improvements to enhance user experience and search accuracy.

Objective

To verify that the mock frontend :

- Accurately interacts with the backend API for creating new satellite configurations.
- Correctly retrieves configurations by mission ID via the *Search* functionality.
- Provides a user-friendly experience with proper error handling and responsiveness.

Scope

This test plan covers :

1. Functional Testing :

- Verification of form inputs and their interaction with the backend API.
- Correct handling of API responses for both the *Create* and *Search* functionalities.

2. Usability Testing :

- Evaluate user experience for input fields, buttons, and overall workflow.

3. Error Handling and Edge Cases :

- Handling invalid data, empty fields, and unexpected input scenarios.

4. Suggestions for Improvement :

- Propose enhancements based on observations.

Test Deliverables

- Test cases for functional, usability, and error scenarios.
- Bug reports.
- Suggested improvements for UI/UX based on findings.

Tools :

- API Testing : Curl, Postman, SoapUI, REST Assured/Java, Robot Framework/Python
- Frontend Testing : Selenium Library in Java/Cucumber/Serenity or in Robot Framework
- Reporting : Allure for results, Serenity or Robot Framework reports

- Functional Testing

Feature : *Create* Functionality	
Test Case 1 : Successful Creation	
Steps :	1. Enter a valid *Configuration Name* (e.g. CubeSat) . 2. Enter a valid *COSPAR ID* (e.g., “2025-001GR”). 3. Enter a valid *Payload Type* (e.g., “OPTICAL”). 4. Click the "*Create*" button.
Expected Result :	• A success response should be logged in the console ("*Success: {...}*"). • A user-friendly success message should appear on the same frontend page. • The data should match the inputs.
Test Case 2 : Empty Required Fields	
Steps :	1. Leave the *Configuration Name* or *COSPAR ID* fields empty. 2. Click the "*Create*" button.
Expected Result :	• Form validation should prevent submission. • A user-friendly error message should appear on the same frontend page.
Test Case 3 : Invalid *COSPAR ID* Format	
Steps :	1. Enter a *COSPAR ID* not matching the “*YYYY-###XX*” pattern (e.g., “INVALID123”). 2. Click the "*Create*" button.
Expected Result :	• An error should be logged in the console indicating invalid *COSPAR ID*. • A user-friendly error message should appear on the same frontend page.
Test Case 4 : Empty *Payload Type*	
Steps :	1. Enter valid *Configuration Name* and *COSPAR ID*. 2. Leave the *Payload Type* field empty. 3. Click the "*Create*" button.
Expected Result :	• Form validation should prevent submission. • A user-friendly error message should appear on the same frontend page.

Comment :	Currently, form submission is possible even with an empty *Payload Type*. However, according to the API design, the *Payload Type* field should be marked as "required" (instead of "optional" as it is currently declared on the *HTML* page). Additionally, its value must be one of the predefined options. To prevent invalid input, we should replace the free-text entry with a dropdown menu or radio buttons for selecting from the predefined *Payload Type* values.
Test Case 5 : Invalid *Payload Type* Value	
Steps :	1. Enter valid *Configuration Name* and *COSPAR ID*. 2. Enter an invalid value in the *Payload Type* field (e.g., “INVALID_TYPE”). 3. Click the "*Create*" button.
Expected Result :	• An error should be logged in the console indicating invalid *Payload Type*. • A user-friendly error message should appear on the same frontend page.
Test Case 6 : Valid *Payload Type* Values	
Steps :	1. Repeat the “*Create*” operation using each of the valid *Payload Type* values : “*OPTICAL*”, “*SAR*”, and “*TELECOM*”.
Expected Result :	• A success response should be logged in the console ("*Success: {...}*"). • A user-friendly success message should appear on the same frontend page. • The data should match the inputs.
Feature : *Search* Functionality	
Test Case 1 : Successful Search	
Steps :	1. Enter a valid Mission *ID* (e.g., “*1*”) in the "*Get Configuration by ID*" field. 2. Click the "*Search*" button.
Expected Result :	• The browser navigates to “*http://localhost:1234/configs/1*”. • The correct configuration details are displayed (based on the mock backend API).
Test Case 2 :	

Non-Existent Mission *ID*	
Steps :	1. Enter a non-existent or invalid *ID* (e.g., “*9999*”). 2. Click the "*Search*" button.
Expected Result :	• The backend returns an error (“*404 Not Found*”). • The browser navigates to “*http://localhost:1234/configs/9999*”. • A user-friendly error message should appear.
Test Case 3 : Empty Mission *ID*	
Steps :	1. Leave the "*Get Configuration by ID*" field empty. 2. Click the "*Search*" button.
Expected Result :	• Form validation should prevent submission. • A user-friendly error message should appear on the same frontend page.
Comment :	Currently, form submission is possible even with an empty *ID*. However, according to the API design, the *ID* field should be marked as "required" (instead of "optional" as it is currently declared on the *HTML* page).

- Usability Testing

Intuitive UI :	• Field labels and placeholders should be clear. • The required fields should be marked or indicated.
Responsiveness :	• The layout should adjust correctly for different screen sizes.
Button States :	• Buttons should be disabled when required fields are empty or invalid.
Dropdown for *Payload Type* :	• Instead of a free-text input for *Payload Type,* a dropdown menu should be used with pre-defined valid options (“*OPTICAL*”, “*SAR*”, “*TELECOM*”) to improve user experience and reduce errors. • Verify accessibility of the dropdown menu for *Payload Type* using keyboard navigation.
Feedback Mechanisms :	• Visual indicators (e.g., loading spinners) for ongoing operations should be present. • Error messages should be clear, consistent, and actionable (e.g., "*Invalid COSPAR ID format. Use YYYY-DDDZZ.*").

- Error Handling and Edge Cases

Input Validation :	• Test alphanumeric, special characters, and excessively long strings (e.g., 500 characters) for all fields. • Test free-text entry (if dropdown is not implemented) to ensure invalid *Payload Type* values are correctly rejected. • Test cases for whitespace-only input or numeric values in the *Payload Type* field. • Test cases where *Payload Type* and *COSPAR ID* are both invalid. Verify error prioritization and messaging. • Test cases where the *Configuration ID* is non-numeric.
Malicious Input Handling :	• Test for *SQL injection* (e.g., *"1 OR 1=1"* in e.g. *COSPAR ID*); the input should be sanitized before being sent to the backend and the backend should reject malicious payloads and log appropriate errors. • Test for *XSS* vulnerabilities by entering *JavaScript* in input fields (e.g., "*<script>alert(1)</script>*" in "*Configuration Name*"); the input should be sanitized before being sent to the backend and the backend should reject malicious payloads and log appropriate errors.
Boundary Testing :	• Test valid *Payload Types* with different capitalizations (e.g., "*optical*", "*Optical*", "*OPTICAL*") to ensure only valid case-sensitive entries are accepted.
Server Error Simulation :	• Simulate API downtime and check user feedback (e.g., "*Error: Unable to connect to server*" in the console).

- Suggestions for Improvement

Error Messaging :	• Display clear, user-friendly error messages on invalid input or server errors (e.g., "*Invalid COSPAR ID format. Please use YYYY-###XX.*").
Real-Time Validation :	• Validate fields like "*COSPAR ID*" in real-time as the user types.
Enhanced Feedback :	• Add visual indicators (e.g., success modals or banners) to inform users of successful operations. • Show success and error messages inline near the fields.
Search Results Display :	• Instead of navigating to a new page, display the retrieved configuration details below the search form for better user experience. • Use an "Accordion Button" to show (and hide) the configuration that has been retrieved.
UI Validation for *Payload Type* :	• Replace free-text entry with a dropdown menu or radio buttons for predefined *Payload Type* values to eliminate invalid input. • According to the API design, all fields are required when creating a configuration. Currently, only the *Configuration Name* and *COSPAR ID* fields are marked as "required" on the frontend. The *Payload Type* field should also be marked as "required" to align with the API design.

UI Validation for Mission *ID* :	• According to the API design, Mission *ID* field is required when searching for a configuration. The Mission *ID* field should also be marked as “required” to align with the API design.
Field Auto-Focus on Error :	• Automatically focus the cursor on the proper field if an error is detected during form submission.
Audit and Logging :	• All actions should be logged in the backend with appropriate details (e.g., timestamp, user ID, action type). • Sensitive data should be excluded from logs. • Audit trail or history view on the frontend to track configuration changes.
Confirmation Dialogs :	• Add confirmation dialogs for critical actions (e.g., delete requests).

- Additional Test Scenarios

Cross-Browser Testing :	• Verify functionality across major browsers (e.g., *Chrome*, *Firefox*, *Safari*, *Edge*).
Accessibility Testing :	• Ensure proper keyboard navigation and focus states; all elements should be accessible to users with disabilities.
Performance Testing :	• Test form submissions under high load (e.g., multiple users creating or searching simultaneously); the frontend should remain responsive and response times should be within acceptable limits. • The API might enforce rate limiting (e.g., *HTTP* error *429 - Too Many Requests* response*)*; the frontend should provide appropriate user feedback.
Access Control Testing : Authentication & Authorization	• Ensure unauthenticated users cannot submit forms for creating and searching configurations. User-friendly messages should appear. • Verify role-specific privileges.
Data Security Testing : Encryption	• Verify sensitive data is transmitted securely; All data transmission uses *HTTPS* and sensitive fields are properly encrypted.
Localization Testing :	• Verify that the UI handles different languages and regional formats.
Negative Testing :	• Test with invalid *HTTP* methods (e.g., *PUT* on search endpoint).
Session Handling :	• Verify behavior when sessions or cookies expire; User should be redirected to a login page or notified to log in again.