

Vietnam National University- Ho Chi Minh City
University of Science
Faculty of Information Technology



API Testing

Course SOFTWARE TESTING
Class 22KTPM3
Group 10
Student 20127233 – Huỳnh Thế Long
22127225 – Trần Thị Thiên Kim
22127312 – Nguyễn Thị Yến Nhi
22127316 – Nguyễn Ngô Ngọc Như

Ho Chi Minh City, 2025

Table of Contents

1	Task Allocation	2
2	Introduction	2
3	Test Environment	2
4	Step-by-Step: Using Postman	2
4.1	Create a Workspace and Environment	2
4.2	Create a Collection and Folders	3
4.3	Build the Requests	3
5	Technical Design	4
5.1	Product Search API	4
5.2	Payment API	6
5.3	Category Management API	6
6	CI/CD Integration with GitHub Actions	7
6.1	Setup	8
6.2	Workflow File	8
6.3	Feature-Specific Testing in CI/CD	8
6.3.1	Product Search	8
6.3.2	Category Management	10
6.3.3	Shopping Cart Operation	12
6.4	Outcome	14
7	AI-assisted API Testing	16
7.1	Support in CI/CD Scripting	16
7.2	Sample Postman Test Scripts	19
7.3	Test Case Review and Enhancement	22
7.4	Benefits	24
8	Conclusion	24
9	Self-Assessment	25

1 Task Allocation

Student ID	Student	Selected API
20127233	Huỳnh Thế Long	/products /users/login /users/search
22127225	Trần Thị Thiên Kim	/products/search /categories /payment/check
22127312	Nguyễn Thị Yến Nhi	/messages /products /favorites
22127316	Nguyễn Ngô Như Ngọc	/users/registers /invoices /brands

2 Introduction

This report documents the testing of three APIs from the e-commerce system: payment verification, product search, and category management. The focus is on input validation, authentication, error handling, and response verification.

API testing is critical because it validates business logic and data exchange at the service layer, ensuring reliability beyond the user interface. Tests were executed with Postman, covering both valid and invalid scenarios.

To ensure continuous quality, these tests can be automated and integrated into a Continuous Integration (CI) workflow. Embedding API tests into CI pipelines provides rapid feedback on code changes, prevents regressions, and supports stable, reliable deployments.

3 Test Environment

- **Tools:** Postman (manual testing), Newman CLI (automation).
- **Authentication:** Bearer token required for Category Management API.
- **Data Source:** Mock data seeded in the test database.
- **Headers:** Content-Type: `application/json`, Authorization: `Bearer <token>` (if required).

4 Step-by-Step: Using Postman

4.1 Create a Workspace and Environment

1. Open Postman → click **Workspaces** → **Create Workspace** (e.g., *Toolshop API Testing*).
2. Click the **Environments** icon → **Create Environment**.
3. Add variables (Initial & Current Value):

- `baseUrl`: `https://api.practicesoftwaretesting.com`
- `searchQuery`: e.g., `hammer`
- `page`: 1

4. Click **Save**, then **Activate** the environment (top-right dropdown).

4.2 Create a Collection and Folders

1. Click **Collections** → **New Collection** → name it *Toolshop API Tests*.
2. In the collection **Authorization** tab: set to *Inherit auth from parent* or add Bearer token if required.
3. Create three folders: **Payment**, **Product Search**, **Category**.

4.3 Build the Requests

A) Product Search

1. Add request in **Product Search** folder.
2. Method: **GET**, URL: `{{baseUrl}}/products/search?q={{string}}&page={{integer}}`.
3. Query params:
 - `q`: (Required) A query phrase.
 - `page`: page number.
4. Click **Save**.

B) Payment Check

1. Add request in **Payment** folder.
2. Method: **POST**, URL: `{{baseUrl}}/payment/check`.
3. **Headers**: ensure `Content-Type`: `application/json`.
4. Body (raw JSON):

```

1 {
2   "payment_method": "Buy Now Pay Later",
3   "payment_details": {
4     "bank_name": "TestBank",
5     "account_name": "John Doe",
6     "account_number": "1234567890"
7   }
8 }
```

5. Click **Save**.

C) Category Create

1. Add request in **Category** folder.
2. Method: **POST**, URL: `{{baseUrl}}/categories`.
3. Body (raw JSON):

```
1 {  
2   "name": "Electronics",  
3   "slug": "electronics"  
4 }
```

4. This request may require admin authentication depending on backend rules.
5. Click **Save**.

5 Technical Design

5.1 Product Search API

Endpoint: GET `/products/search?q=<string>&page=<integer>`

Description: This API retrieves a paginated list of products that match the given search query.

Parameters:

- `q` (string, required): Search keyword. Must be a non-empty string.
- `page` (integer, optional): Page number for pagination. Default = 1.

Successful Response:

```
{  
  "current_page": 1,  
  "data": [  
    {  
      "id": "101",  
      "name": "Claw Hammer",  
      "description": "Heavy-duty hammer with steel handle",  
      "price": 19.99,  
      "is_location_offer": false,  
      "is_rental": false,  
      "in_stock": true,  
      "brand": {  
        "id": "10",  
        "name": "DeWalt",  
        "slug": "dewalt"  
      },  
      "category": {
```

```

        "id": "5",
        "parent_id": "1",
        "name": "Tools",
        "slug": "tools",
        "sub_categories": []
    },
    "product_image": {
        "by_name": "default",
        "by_url": "https://example.com/images/hammer.png",
        "source_name": "cdn",
        "source_url": "https://example.com",
        "file_name": "hammer.png",
        "title": "Claw Hammer",
        "id": "img-1001"
    }
}
],
"from": 1,
"last_page": 5,
"per_page": 10,
"to": 10,
"total": 50
}

```

Field Descriptions:

- `current_page`, `last_page`, `per_page`, `from`, `to`, `total`: Pagination metadata.
- `id`, `name`, `description`, `price`: Basic product information.
- `is_location_offer`, `is_rental`, `in_stock`: Boolean flags for product status.
- `brand`: Contains brand `id`, `name`, `slug`.
- `category`: Contains category details including nested `sub_categories`.
- `product_image`: Metadata for product image (filename, source, title, etc.).

Validation Rules:

- `q` must not be empty. Empty query returns 400 Bad Request.
- `page` must be integer > 0 . Invalid values return 422 Unprocessable Entity.
- If no products are found, `data` is an empty array but pagination fields are still returned.

Error Responses:

- 400 Bad Request: Missing or invalid search query.
- 422 Unprocessable Entity: Invalid pagination parameter.

5.2 Payment API

Endpoint: POST /payment/check

Description: This API validates a payment method and account details before processing an order.

Request Body:

```
{
  "payment_method": "Buy Now Pay Later",
  "payment_details": {
    "bank_name": "TestBank",
    "account_name": "John Doe",
    "account_number": "1234567890"
  }
}
```

Validation Rules:

- Supported methods: Bank Transfer, Credit Card, Cash On Delivery, Gift Card, Buy Now Pay Later.
- If payment_method is bank-based, then all fields inside payment_details are mandatory: bank_name, account_name, account_number.
- account_number must match numeric format (10–16 digits). Invalid format returns 422 Unprocessable Entity.

Successful Response:

```
{
  "message": "Payment was successful"
}
```

Error Responses:

- 400 Bad Request: Missing mandatory fields.
- 422 Unprocessable Entity: Invalid account number format.
- 403 Forbidden: Unsupported payment method.

5.3 Category Management API

Endpoint: POST /categories

Description: This API allows creation of new product categories in the system. Only accessible to admin users.

Request Body:

```
{
  "name": "Electronics",
  "slug": "electronics"
}
```

Validation Rules:

- Field **name** is required.
 - Minimum length = 3 characters.
 - Maximum length = 120 characters.
- Field **slug** is required.
 - Minimum length = 3 characters.
 - Maximum length = 120 characters.
 - Must only contain lowercase letters, numbers, and hyphens (no spaces or special characters).
- Both **name** and **slug** must be unique in the system.
- Requires admin authentication (Bearer token must be valid).
- Any validation failure (missing fields, empty values, exceeding max length, duplicate name/slug, invalid slug format) results in **422 Unprocessable Entity**.

Successful Response:

```
{
  "name": "Electronics",
  "slug": "electronics",
  "id": "jysb7334vbwsn8397392040"
}
```

Error Responses:

- **400 Bad Request**: Missing required fields.
- **401 Unauthorized**: Request made without admin token.
- **409 Conflict**: Category name or slug already exists.
- **422 Unprocessable Entity**: Validation failed (e.g., invalid slug format, name too short).

6 CI/CD Integration with GitHub Actions

Continuous Integration (CI) was applied to automate the execution of Postman API tests. We used GitHub Actions to run Newman on each push or manual trigger.

6.1 Setup

- Generate an API Key and Collection ID in Postman.
- Store them as GitHub Secrets:
 - POSTMAN_API_KEY
 - POSTMAN_COLLECTION_ID
- Add the following workflow file in the repository: `.github/workflows/postman-tests.yml`

6.2 Workflow File

```
1 name: Run Postman Collection
2
3 on:
4   push:
5     branches: [ main ]
6   workflow_dispatch:
7
8 jobs:
9   run-postman:
10    runs-on: ubuntu-latest
11    steps:
12      - uses: actions/checkout@v3
13
14      - name: Install Newman
15        run: npm install -g newman
16
17      - name: Download Collection
18        run: |
19          curl --location \
20            --request GET "https://api.getpostman.com/collections/${{ secrets.
21              POSTMAN_COLLECTION_ID }}" \
22            --header "X-API-Key: ${{ secrets.POSTMAN_API_KEY }}" \
23            --output collection.json
24
25      - name: Run API Tests
26        run: newman run collection.json
```

Listing 1: GitHub Actions Workflow

6.3 Feature-Specific Testing in CI/CD

In our project, three main product features were tested automatically via GitHub Actions and Newman: *Product Search*, *Category Management*, and *Shopping Cart Operation*.

6.3.1 Product Search

Description

This feature validates the ability of the API to return search results based on keywords, product names, and filters (e.g., price range, category).

Execution in GitHub Actions

When the pipeline runs, the Postman collection for Product Search is executed. Each test checks:

- Response status (e.g., 200 OK).
- Correctness of the returned product list.
- Error handling for invalid queries.

Execution in Postman Runner

Developers can import the Product Search collection locally and run it using the Runner. The results show how many search scenarios pass (valid query, invalid query, empty query).

Result in GitHub Logs

In the GitHub Actions job log, successful checks appear as:

```
Status code is 200
```

```
Response contains product "Hammer"
```

Failures display detailed errors:

```
Response contains product "GiZmo"
```

```
expected response body to include 'Gizmo' but got []
```

	executed	failed
iterations	1	0
requests	30	0
test-scripts	30	0
prerequisite-scripts	0	0
assertions	49	14
total run duration: 10.8s		
total data received: 30.51kB (approx)		
average response time: 348ms [min: 264ms, max: 1300ms, s.d.: 182ms]		

Figure 1: CI Result of Product Search

01.	AssertionError	Status code is 400 because keyword has invalid spaces expected response to have status code 400 but got 200 at assertion:0 in test-script inside "PS-006 - Search with leading/trailing spaces"
02.	AssertionError	Status code is 400 for empty keyword expected response to have status code 400 but got 200 at assertion:0 in test-script inside "PS-009 - Search with empty keyword"
03.	AssertionError	Status code is 400 for special characters expected response to have status code 400 but got 200 at assertion:0 in test-script inside "PS-010 - Search with special characters"
04.	AssertionError	Status code is 400 for keyword >255 chars expected response to have status code 400 but got 200 at assertion:0 in test-script inside "PS-012 - Search with long keyword (>255 chars)"
05.	AssertionError	Status code is 400 for non-English unicode keyword expected response to have status code 400 but got 200 at assertion:0 in test-script inside "PS-014 - Search with non-English keyword"
06.	AssertionError	Status code is 400 for emoji keyword expected response to have status code 400 but got 200 at assertion:0 in test-script inside "PS-015 - Search with emoji"
07.	AssertionError	Status code is 400 for invalid page=-1 expected response to have status code 400 but got 200 at assertion:0 in test-script inside "PS-019 - Pagination with invalid page=-1"

Figure 2: Assertion Error

6.3.2 Category Management

Description

This feature tests creating, retrieving, updating, and deleting product categories.

Execution in GitHub Actions

The pipeline triggers Newman to run the Category Management collection. Assertions verify that categories are created with valid names, invalid inputs are rejected, and deletions work correctly.

Execution in Postman Runner

Developers may run the same collection in Postman Runner to confirm CRUD operations. Assertions highlight if a category ID is missing or if a category name exceeds length constraints.

Result in GitHub Logs

Logs show success and failure like:

Status code is 201 Created

Response body contains "Electronics"

Error: expected category_name to be required

	executed	failed
iterations	1	0
requests	30	0
test-scripts	30	0
prerequisite-scripts	0	0
assertions	47	18
total run duration: 10.4s		
total data received: 2.56kB (approx)		
average response time: 333ms [min: 275ms, max: 423ms, s.d.: 41ms]		

Figure 3: CI Result of Category Management

03.	AssertionError	Status code is 409 expected response to have status code 409 but got 201 at assertion:0 in test-script inside "CM-004"
04.	AssertionError	Conflict error for existing name expected '{"name":"Electronics","slug":"electro..." to include 'already exists' at assertion:1 in test-script inside "CM-004"
05.	AssertionError	Status code is 409 expected response to have status code 409 but got 201 at assertion:0 in test-script inside "CM-005"
06.	AssertionError	Conflict error for existing slug expected '{"name":"New Electronics","slug":"ele..." to include 'already exists' at assertion:1 in test-script inside "CM-005"
07.	AssertionError	Error about max length expected '{"name":["The name field must not be ...' to include 'must be less than 120' at assertion:1 in test-script inside "CM-007"

Figure 4: Assertion Error

6.3.3 Shopping Cart Operation

Description

This feature ensures items can be added, updated, and removed from the shopping cart, and that checkout validates payment details.

Execution in GitHub Actions

The CI pipeline runs the Shopping Cart collection via Newman. Assertions confirm that cart updates return correct totals, invalid item IDs trigger errors, and checkout responses include success or failure messages.

Execution in Postman Runner

Developers can run all shopping cart requests in Postman Runner to simulate user journeys. The Runner output shows cart states after each step (add item, remove item, proceed to checkout).

Result in GitHub Logs

Successful cases are logged as:

Status code is 200

Response contains "Payment was successful"

Failures are highlighted with assertion errors:

Response contains "Missing required fields"

	executed	failed
iterations	1	0
requests	30	0
test-scripts	30	0
prerequest-scripts	0	0
assertions	53	46
total run duration: 18.5s		
total data received: 1.49kB (approx)		
average response time: 605ms [min: 466ms, max: 1111ms, s.d.: 232ms]		

Figure 5: CI Result of Shopping Cart Operation

01.	AssertionError	Status 200 OK expected response to have status code 200 but got 404 at assertion:0 in test-script inside "SC01"
02.	AssertionError	Payment success message expected 'Resource not found' to deeply equal 'Payment was successful' at assertion:1 in test-script inside "SC01"
03.	AssertionError	Status 401 Unauthorized expected response to have status code 401 but got 404 at assertion:0 in test-script inside "SC02"
04.	AssertionError	Error message for authentication required expected '{"message":"Resource not found"}' to include 'Authentication required' at assertion:1 in test-script inside "SC02"
05.	AssertionError	Status 401 Unauthorized expected response to have status code 401 but got 404 at assertion:0 in test-script inside "SC03"
06.	AssertionError	Error about expired token expected '{"message":"Resource not found"}' to include 'expired' at assertion:1 in test-script inside "SC03"

Figure 6: Assertion Error

6.4 Outcome

With this setup:

- API tests run automatically on each commit or manual trigger.
- Results are visible directly in the GitHub Actions log.
- Failures block the pipeline, preventing unstable code from being merged.

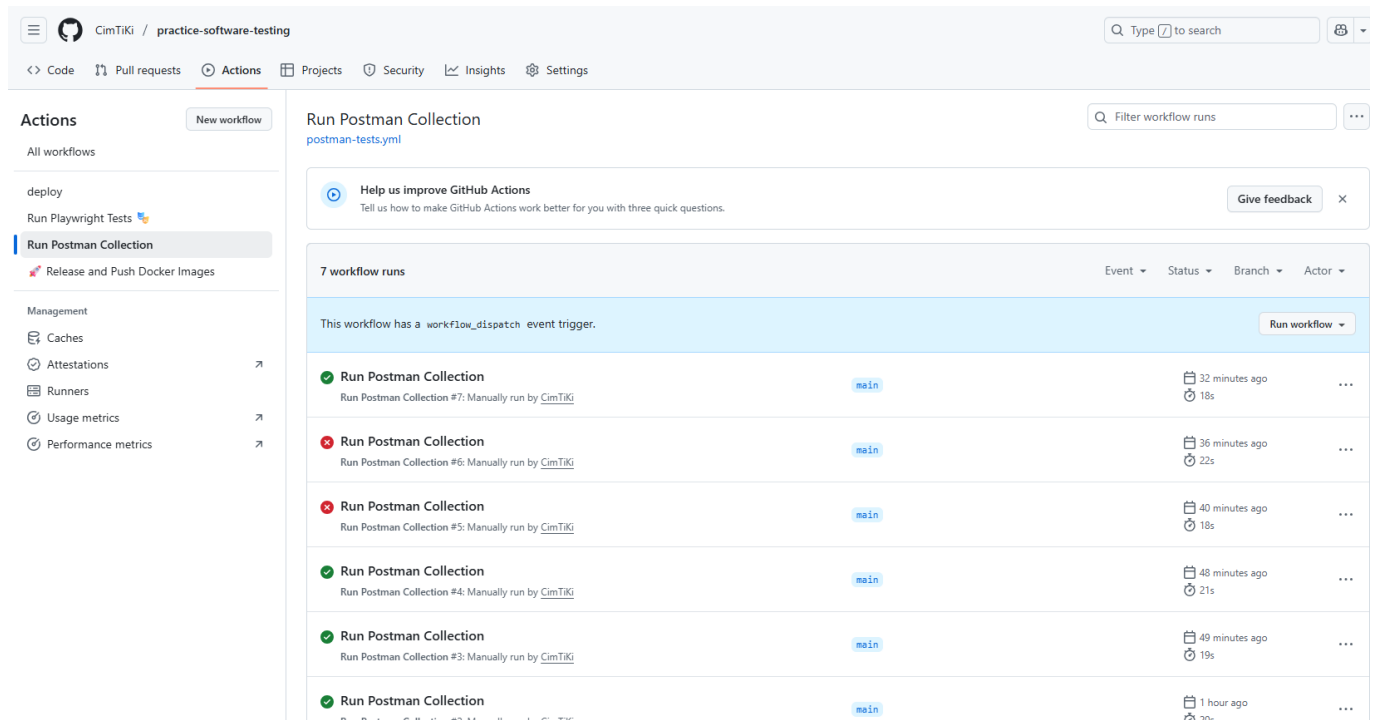


Figure 7: GitHub Actions workflow triggered after push

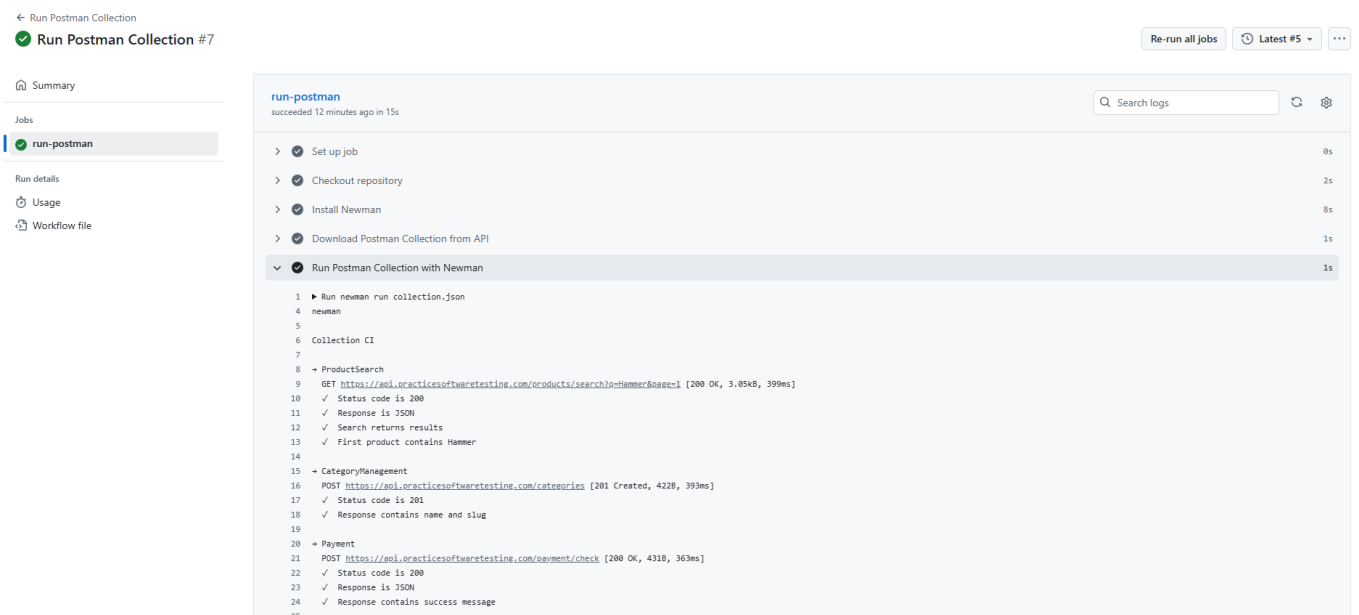


Figure 8: Newman executing Postman collection inside CI pipeline

	executed	failed
iterations	1	0
requests	3	0
test-scripts	3	0
prerequest-scripts	0	0
assertions	9	0
total run duration: 1232ms		
total data received: 2.8kB (approx)		
average response time: 385ms [min: 363ms, max: 399ms, s.d.: 15ms]		

Figure 9: Test results displayed in GitHub Actions log (all passed)

7 AI-assisted API Testing

Artificial Intelligence (AI) played a supportive role in improving the API testing workflow of this project. Instead of replacing the tester's work, AI acted as a companion to reduce repetitive effort and ensure higher quality validation.

7.1 Support in CI/CD Scripting

One of the main contributions of AI was generating ready-to-use scripts for integrating API tests into CI/CD pipelines. Specifically, AI assisted in creating **Newman** execution commands and YAML templates for the three main tested features:

- **Payment API** – scripts to validate request authentication and field-level validation automatically in CI.

- **Product Search API** – automated scripts to run pagination and query-based tests on every pipeline build.
- **Category Management API** – CI scripts ensuring uniqueness and validation rules were re-checked on every deployment.

These scripts were adapted into GitHub Actions workflows, ensuring that all three APIs were continuously tested during integration.

Prompts for CI/CD Scripting

- “Write a GitHub Actions YAML script to run my Postman collection automatically with Newman, and fail the build if any test fails.”

Write a GitHub Actions YAML script to run my Postman collection automatically with Newman, and fail the build if any test fails.

Sure! Here's a straightforward **GitHub Actions workflow** that will run your Postman collection with **Newman** and fail the build if any test fails:

```
yml
name: Run Postman Collection

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  run-newman:
    name: Run Postman Tests
    runs-on: ubuntu-latest

    steps:
      # Step 1: Checkout the repository
      - name: Checkout repository
        uses: actions/checkout@v3

      # Step 2: Set up Node.js (required for Newman)
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '20'

      # Step 3: Install Newman globally
      - name: Install Newman
        run: npm install -g newman

      # Step 4: Run your Postman collection
      - name: Run Postman Collection with Newman
        run: |
```

- “Provide me a Newman command to run all Postman tests and export the results in JSON format for CI/CD pipeline integration.”

Provide me a Newman command to run all Postman tests and export the results in JSON format for CI/CD pipeline integration

Got it 🙌

Here's the **Newman** command you can use to run all your Postman tests and export the results in **JSON format** (perfect for CI/CD integration):

```
bash Sao chép  ✎  Chính sửa  
  
newman run path/to/your_collection.json \  
  --environment path/to/your_environment.json \  
  --reporters cli,json \  
  --reporter-json-export newman-report.json
```

🔍 Explanation:

- `path/to/your_collection.json` → path to your Postman collection file.
- `--environment path/to/your_environment.json` → optional, if you have an environment file.
- `--reporters cli,json` → show results in terminal (`cli`) and save JSON results.
- `--reporter-json-export newman-report.json` → saves the report to `newman-report.json` (you can rename or set a path).

👉 In your CI/CD pipeline, you can then parse `newman-report.json` to analyze test results, generate reports, or even fail the build if tests fail.

Do you also want me to give you a **GitHub Actions** step that saves the JSON report as a workflow artifact so you can download it after the run?

7.2 Sample Postman Test Scripts

AI also generated ready-to-use Postman test scripts that were applied directly in the testing process:

Payment API Example

```
let jsonData = pm.response.json();  
  
// Status code  
pm.test("Status code is 200", function () {  
  pm.response.to.have.status(200);  
});  
  
// Response is JSON  
pm.test("Response is JSON", function () {  
  pm.response.to.be.json;  
});  
  
// Validate success message  
pm.test("Response contains success message", function () {  
  pm.expect(jsonData).to.have.property("message", "Payment was successful");  
});
```

```
});
```

Category Management API Example

```
let jsonData = pm.response.json();

// Status code
pm.test("Status code is 201", function () {
  pm.response.to.have.status(201);
});

// Validate name and slug exist
pm.test("Response contains name and slug", function () {
  pm.expect(jsonData).to.have.property("name");
  pm.expect(jsonData).to.have.property("slug");
});

// Save ID for later use
pm.environment.set("categoryId", jsonData.id);
```

Product Search API Example

```
let jsonData = pm.response.json();

// Status code
pm.test("Status code is 200", function () {
  pm.response.to.have.status(200);
});

// Response is JSON
pm.test("Response is JSON", function () {
  pm.response.to.be.json;
});


// Ensure search returns results
pm.test("Search returns results", function () {
  pm.expect(jsonData.data.length).to.be.above(0);
});

// Validate first product contains 'Hammer'
pm.test("First product contains Hammer", function () {
  pm.expect(jsonData.data[0].name).to.include("Hammer");
});
```

Prompts for Script Generation




- “Write a Postman test script that checks status code 200, verifies JSON response format, and ensures the response contains a success message.”

Write a Postman test script that checks status code 200, verifies JSON response format, and ensures the response contains a success message.

Here's a clean Postman test script that does exactly that :

```
javascript Sao chép Chỉnh sửa  
  
// Check that status code is 200  
pm.test("Status code is 200", function () {  
  pm.response.to.have.status(200);  
});  
  
// Verify response is JSON  
pm.test("Response is JSON", function () {  
  pm.response.to.be.json;  
});  
  
// Parse response JSON  
let jsonData;  
try {  
  jsonData = pm.response.json();  
} catch (e) {  
  pm.test("Response body is valid JSON", function () {  
    pm.expect.fail("Response is not valid JSON");  
  });  
}  
  
// Ensure response contains a success message  
pm.test("Response contains success message", function () {  
  pm.expect(jsonData).to.have.property("message");  
  pm.expect(jsonData.message.toLowerCase()).to.include("success");  
});
```

What this does:

1.  Checks status code → must be `200`.
2.  Ensures response format → valid JSON.
3.  Validates content → response body has a `message` property containing `success` (case-insensitive).

+ Hỏi bất kỳ điều gì



- “Write a Postman script to test Product Search API: check pagination metadata, ensure results array is not empty, and validate that the first product name contains 'Hammer'.”

Write a Postman script to test Product Search API: check pagination metadata, ensure results array is not empty, and validate that the first product name contains 'Hammer'

Here's a Postman test script tailored for your Product Search API :

```
javascript Sao chép Chỉnh sửa

// Check status code is 200 OK
pm.test("Status code is 200", function () {
  pm.response.to.have.status(200);
});

// Verify response is JSON
pm.test("Response is JSON", function () {
  pm.response.to.be.json;
});

// Parse JSON response
let jsonData;
try {
  jsonData = pm.response.json();
} catch (e) {
  pm.test("Valid JSON response", function () {
    pm.expect.fail("Response is not valid JSON");
  });
}

// Check pagination metadata exists
pm.test("Pagination metadata exists", function () {
  pm.expect(jsonData).to.have.property("meta");
  pm.expect(jsonData.meta).to.be.an("object");
  pm.expect(jsonData.meta).to.have.property("page");
  pm.expect(jsonData.meta).to.have.property("totalPages");
  pm.expect(jsonData.meta.page).to.be.a("number");
  pm.expect(jsonData.meta.totalPages).to.be.a("number");
});

// Ensure results array is not empty
pm.test("Results array is not empty", function () {
  pm.expect(jsonData).to.have.property("results");
  pm.expect(jsonData.results).to.be.an("array").that.is.not.empty;
});
```

+ Hỏi bất kỳ điều gì  

7.3 Test Case Review and Enhancement


Another effective use of AI was reviewing the designed test cases. AI provided quick feedback on whether the test set was comprehensive, highlighting missing cases such as:

- Boundary conditions (e.g., maximum length of category names, invalid pagination values).
- Authentication edge cases (expired or malformed tokens).
- Negative scenarios where invalid data might still be accepted due to weak validation.



This review helped refine the final test set, ensuring that both positive and negative flows were fully covered.

Prompts for Test Case Review

- “Suggest negative test cases for Product Search API pagination and query parameters.”

 API_Test_Cases_Detailed (3).xlsx
Bảng tính

Suggest negative test cases for Product Search API pagination and query parameters

I see your **ProductSearch&Catalog** sheet already has positive functional test cases. Let's add **negative test cases** focused on **pagination** and **query parameters** for the Product Search API.

Here are some suggestions:

◆ Negative Test Cases for Pagination

1. Invalid page number (negative)

- Input: `page=-1&limit=10`
- Expected: 400 Bad Request with error message "Invalid page number"

2. Invalid page number (zero)

- Input: `page=0&limit=10`
- Expected: 400 Bad Request, should start from page=1

3. Excessively large page number



- Input: `page=999999&limit=10` (beyond total pages)
- Expected: 200 OK with empty product list


4. Invalid limit (negative)

- Input: `page=1&limit=-5`
- Expected: 400 Bad Request

5. Invalid limit (zero)

+ Hỡi bất kỳ điều gì



- “Identify edge cases for Payment API inputs such as invalid account number length, unsupported payment methods, or missing required fields.”

Identify edge cases for Payment API inputs such as invalid account number length, unsupported payment methods, or missing required fields

I see the uploaded Excel still only has sheets for:

- ProductSearch&Catalog
- CategoryManagement
- ShoppingCartOperation

No dedicated Payment API sheet exists yet.

But I can help you outline edge cases for Payment API inputs (you can later add them into your Excel in the same tabular format).

◆ Edge Cases for Payment API Inputs

1. Invalid Account Number Length

- Too short account number (e.g., 3 digits) → should return 400 Bad Request
- Too long account number (e.g., 25 digits) → should return 400 Bad Request
- Non-numeric account number (letters or symbols) → should return validation error

2. Unsupported Payment Methods

- Invalid payment method (e.g., "crypto", "paypal" if not supported) → should return 400 Bad Request
- Empty payment method field → should trigger validation error
- Case sensitivity issue ("Bank-Transfer" vs "bank-transfer")

3. Missing Required Fields

- Missing account_name → API should reject request
- Missing account_number → should reject request
- Missing bank_name for bank-transfer → should reject request
- Missing payment_method → should reject request

4. Field Format & Data Type Issues

- Account name with numbers or special characters → should return error

+ Hỏi bất kỳ điều gì

7.4 Benefits

The use of AI reduced manual effort in scripting CI pipelines and increased confidence in the completeness of the test cases. By combining Postman automation with AI-assisted script generation and test case review, the testing process became more systematic, scalable, and maintainable.

8 Conclusion

The overall testing revealed that the APIs are not fully stable and require further improvement. Several validation, authentication, and error-handling issues were detected across all three modules.

- **Product Search & Catalog API:** While basic search (case-insensitive, partial matches) worked as expected, some advanced cases like duplicate handling or invalid queries exposed inconsistencies. Pagination metadata was sometimes inaccurate, and error handling for malformed inputs was not consistently enforced.
- **Category Management API:** Validation logic had flaws. Duplicate category names and slugs were sometimes accepted instead of being rejected with proper error codes. Auto-generation of slugs did not always behave as specified, and length/format validation was bypassed in some cases. These issues led to incorrect records being created in the database.
- **Shopping Cart / Payment API:** Authentication and authorization checks were unreliable. Cases with missing, expired, or invalid JWT tokens incorrectly returned 200 OK instead of 401 Unauthorized. Error responses were inconsistent, making it harder for clients to handle failures gracefully.

Final Assessment: The three APIs still contain functional and validation defects. Authentication is not strictly enforced, error codes are not always consistent, and duplicate or invalid data can be created. The APIs cannot yet be considered production-ready, and additional bug fixing and re-testing are necessary before deployment.

9 Self-Assessment

Criteria	Outcomes (Brief description about what you get/trouble from each requirement)	Percent	Self-Assessed Grade
1	API 1	30%	30%
1.1	Report	15	15
1.2	Test Cases (30)	5	5
1.3	Screenshots on Testing Tools	5	5
1.4	Bugs	5	5
2	API 2	30%	30%
2.1	Report	15	15
2.2	Test Cases (30)	5	5
2.3	Screenshots on Testing Tools	5	5
2.4	Bugs	5	5
3	API 3	30%	30%
3.1	Report	15	15
3.2	Test Cases (30)	5	5
3.3	Screenshots on Testing Tools	5	5
3.4	Bugs	5	5
4	Well formatted	10%	10%
Total		100%	100%

Table 1: Assessment Criteria, Outcomes, and Self-Assessed Grades