Gator Market

SW Engineering CSC 648/848
Spring 2025

TEAM 4

Team Lead: Clark Batungbakal
cbatungbakal@sfsu.edu
Front-end Lead: Thanh Duong
Front-end member: Daniel Lee
Back-end Lead: Caleb Onuonga
GitHub Master: Trinity Godwin

Version	Date Submitted	Revised
Ver 1	03/12/2025	03/21/2025
Ver 2	03/25/2025	04/03/2025
Ver 3	04/25/2025	04/27/2025
Ver 4	05/16/2025	05/20/2025

TABLE OF CONTENTS

PRODUCT SUMMARY 3
USABILITY TEST PLAN - SEARCH FUNCTION 4
QUALITY ASSURANCE TEST PLAN AND TESTING 6
PEER CODE REVIEW 8
BEST PRACTICES FOR SECURITY 17
ADHERENCE TO ORIGINAL NON-FUNCTIONAL REQUIREMENTS 18

PRODUCT SUMMARY

Name of the product?

GatorMarket

Product Description

GatorMarket is a student-exclusive online marketplace built for the San Francisco State University (SFSU) community. It provides a safe and convenient platform where students can buy, sell, and trade items directly with one another. GatorMarket makes it easy to find or list what students need within a trusted network, from textbooks and electronics to furniture and school supplies. Unlike public platforms like Craigslist or Facebook Marketplace, GatorMarket verifies users through their @sfsu.edu email address, ensuring all members are part of the SFSU community. This adds a layer of trust and campus relevance that external platforms lack.

The application is designed with simplicity in mind. Students can post listings with images and descriptions, search by keyword or category, and contact sellers directly through a secure messaging system. GatorMarket reduces friction in student transactions by providing an all-in-one interface that is tailored to the university's needs, helping students save money, declutter, and connect locally.

Final Priority Features

These are the core functions the GatorMarket team is committed to fully delivering and testing for the final release:

- Users can register using a valid @sfsu.edu email address.
- Users must accept the terms of service during registration.
- Passwords are securely encrypted before storage in the database.
- Users can securely log in and log out.
- Logged-in users can create new listings with:
    - Title
    - Description
    - Image upload
    - Category
    - Price
- Listings are displayed in a browsable, responsive grid layout.
- Users can filter listings by category or keyword.
- Users can search listings using a search bar (up to 40 alphanumeric characters).
- Users can view detailed information for individual listings.
- The website is accessible and mobile-friendly.

Note: While our posting and messaging features are designed and partially implemented, they are not currently functioning end-to-end due to unresolved issues with data persistence and live communication. We've identified where the breakdown occurs and plan to continue debugging after the final demo.

USABILITY TEST PLAN - SEARCH FUNCTION

Test Objectives

The search function is a major part of our site. Every user, buyer, and seller will use it. The first thing the user will see is the home page with a preview of our products. If a user is looking for something specific, they can use the search function to find it quickly. We have a categorical search and a string search. Categorical for a broad search and string for anything specific.

Categorical:
- Electronics
- Books
- Fashion
- Home and Kitchen
- Computers and Accessories

Test Background and Setup

System Setup: Any device with an internet connection and a modern browser.

Starting Point: http://13.52.231.140:3001/ (Homepage of GatorMarket)

HW Tester Needs: Device with Chrome, Firefox, Safari, etc.

Intended Users: SFSU students and staff

Test Environment: Conducted on campus, in person, without prior training. Monitor starts off; time begins when monitor is turned on.

URL: http://13.52.231.140:3001/

Usability Task Description

Task: Search for a book using either categorical or string-based search.
Starting Condition: User begins on homepage.
Success Criteria: User finds the book using either method.
Benchmark: Task completed in under 15 seconds.

Effectiveness Evaluation

The search function is evaluated by how easily a user can locate a specific item (in this case, a book) using either method. Our internal testing showed that it takes under 5 seconds for a familiar user. To account for new users, we used Carnegie Mellon's Eberly Center guidance, allowing ~15 seconds for task completion. Observers will note whether the user prefers categorical or string search, and if any confusion arises during the process.

Efficiency Evaluation

Efficiency will be measured by how long it takes a user to complete the task from a cold start (monitor off). Once the monitor is turned on and the homepage loads, the timer begins. Time will be recorded once the correct result is found, and the method used (categorical or string) will be documented.

User Satisfaction Evaluation (Likert Scale Evaluation)

Based on a scale of: Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree

a) The search bar was easy to find

b) The categorical search was useful

c) The string search was useful

GENAI Usage

Tool Used: ChatGPT (GPT-4)

How We Used It:
ChatGPT was used to review and improve clarity in our task instructions, Likert scale questions, and evaluation criteria. It also helped us rephrase technical terms to better suit non-technical users and align our test logic with usability testing best practices.

Prompts Used:

- “Can you review this usability test description for clarity?”

- “Is this Likert scale well-formatted for a usability test?”

- “How can I better define efficiency and effectiveness in a test plan?”

Key Suggestions from GenAI:

- Recommended separating categorical and string examples more clearly

- Improved the phrasing of success criteria

- Suggested reordering instructions for better flow

Utility Ranking: HIGH

QUALITY ASSURANCE TEST PLAN AND TESTING

Test Objectives

Evaluating the functionality, accuracy, and reliability of the search feature, ensuring it returns correct and relevant results based on user input. The QA process also validates how the search behaves with both valid and invalid inputs.

Hardware and Software Setup

Hardware: Desktop, laptop, tablet, or smartphone with stable internet connection.

Software: Chrome v118 and Firefox v117 browsers

Test Environment: Postman for API testing

URL: http://13.52.231.140:3001

Feature to be Tested

The search functionality allows users to:

1. Search for listings by entering text in the search bar (matches against product titles and descriptions)
2. Filter listings by category using a dropdown menu
3. Combine text search with category filtering
4. View matching results with relevant product information

QA Test Plan

Test ID	Test Title	Test Description (What is being tested)	Test Input	Expected Correct Output	Chrome v118	Firefox v117
S-001	Basic Text Search	Test if the system returns correct results when searching by keyword	Navigate to homepage, enter “table” in search bar, click search button	Status 200, JSON response with listings containing “table” in title or description, count > 0	PASS	PASS
S-002	Category Filter	Test if the system filter results correctly by category	Navigate to homepage, select “Books” (category 1) from dropdown, click search button	Status 200, JSON response with listings only from Books category, count > 0	PASS	PASS
S-003	Combined Search	Test if the system correctly combines text search with category filtering	Navigate to homepage, select “Furniture” (category 3) from dropdown, enter “table” in search bar, click search button	Status 200, JSON response with listings containing “table” in title/decription AND from Furniture category only	PASS	PASS

Test Execution Results

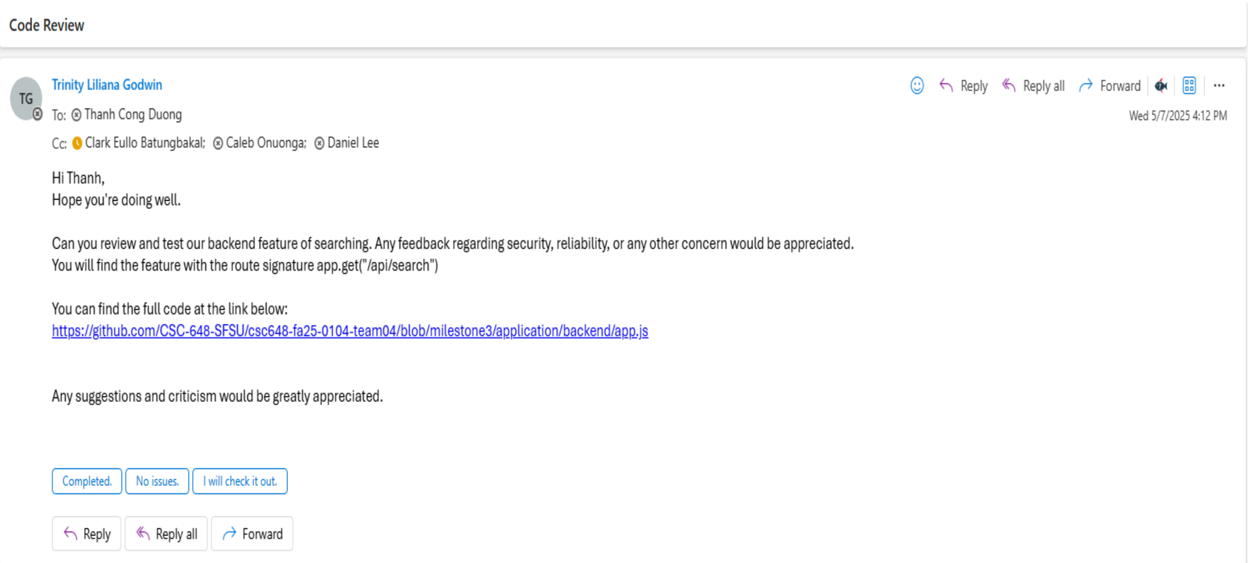
All tests were successfully executed on both Chrome v118 and Firefox v117 browsers. The search functionality performed as expected, with proper filtering based on text queries and category selection. Response times were consistently fast (under 500ms), and the user interface correctly displayed the filtered results.
No significant differences were observed between the two browsers. The search functionality handled all test cases appropriately and returned the expected results in both environments.

GenAI Usage

GenAI Tool Used: Claude 3.7 Sonnet
How the Tool Was Used: The tool was used to analyze our search functionality code and generate
a structured QA test plan that focuses on the key functionality. It helped identify critical test cases
and expected behaviors.
Key Examples and Prompts: We provided the codebase for our search functionality API and asked for a QA test plan with 3 essential test cases focusing on text search, category filtering, and
combined search. The prompt specified the need for browser compatibility testing. Utility Ranking: HIGH - The GenAI tool efficiently analyzed our code and produced a comprehensive test plan aligned with software engineering best practices, saving significant time in test planning while ensuring thorough coverage of the functionality.

PEER CODE REVIEW

Code Review E-mail



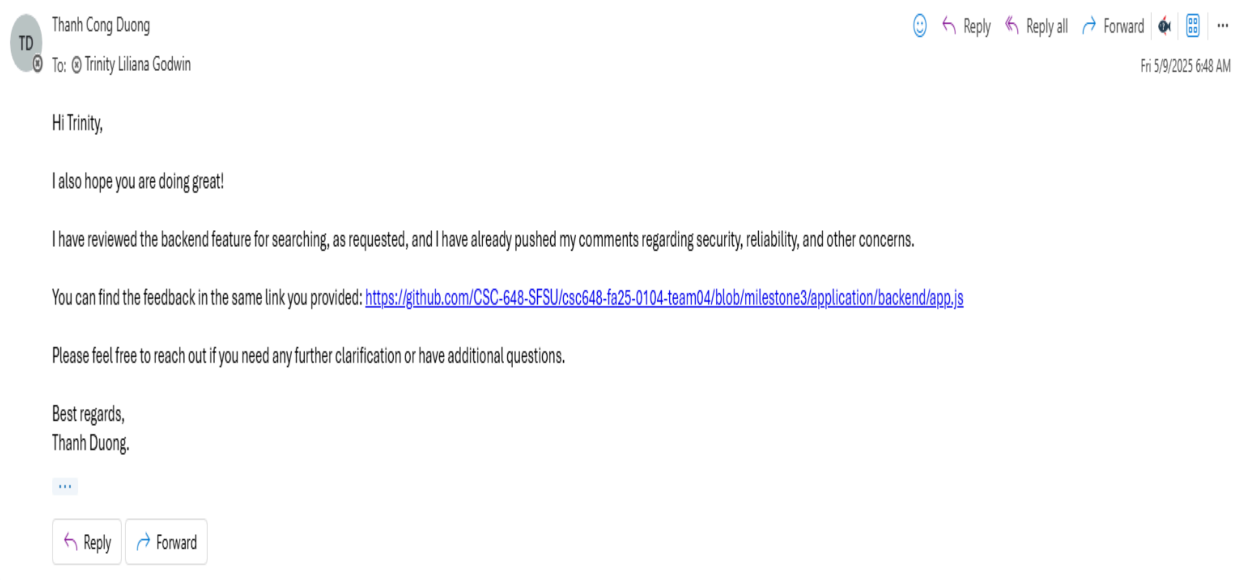
Trinity Liliana Godwin
To: Thanh Cong Duong
Cc: Clark Eullo Batungbakal; Caleb Onuonga; Daniel Lee
Wed 5/7/2025 4:12 PM

Hi Thanh,
Hope you're doing well.

Can you review and test our backend feature of searching. Any feedback regarding security, reliability, or any other concern would be appreciated.
You will find the feature with the route signature app.get("/api/search")

You can find the full code at the link below:
https://github.com/CSC-648-SFSU/csc648-fa25-0104-team04/blob/milestone3/application/backend/app.js

Any suggestions and criticism would be greatly appreciated.



Thanh Cong Duong
To: Trinity Liliana Godwin
Fri 5/9/2025 6:48 AM

Hi Trinity,

I also hope you are doing great!

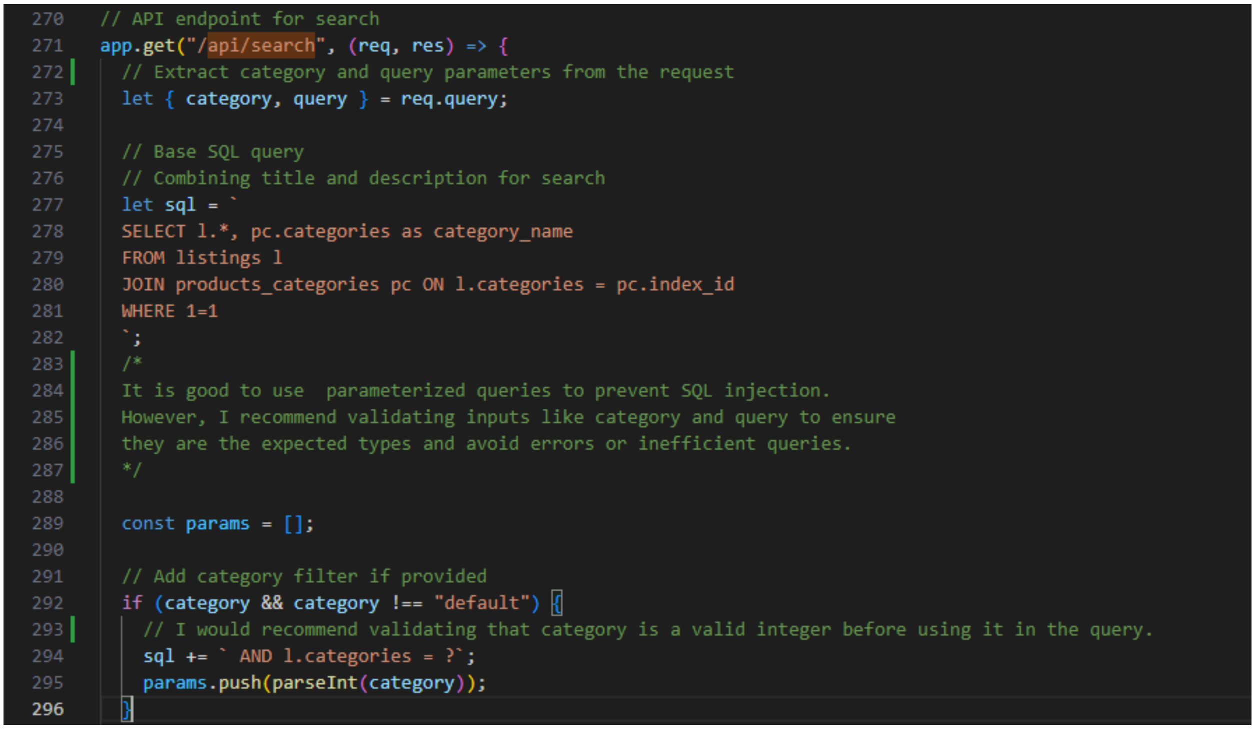
I have reviewed the backend feature for searching, as requested, and I have already pushed my comments regarding security, reliability, and other concerns.

You can find the feedback in the same link you provided: https://github.com/CSC-648-SFSU/csc648-fa25-0104-team04/blob/milestone3/application/backend/app.js

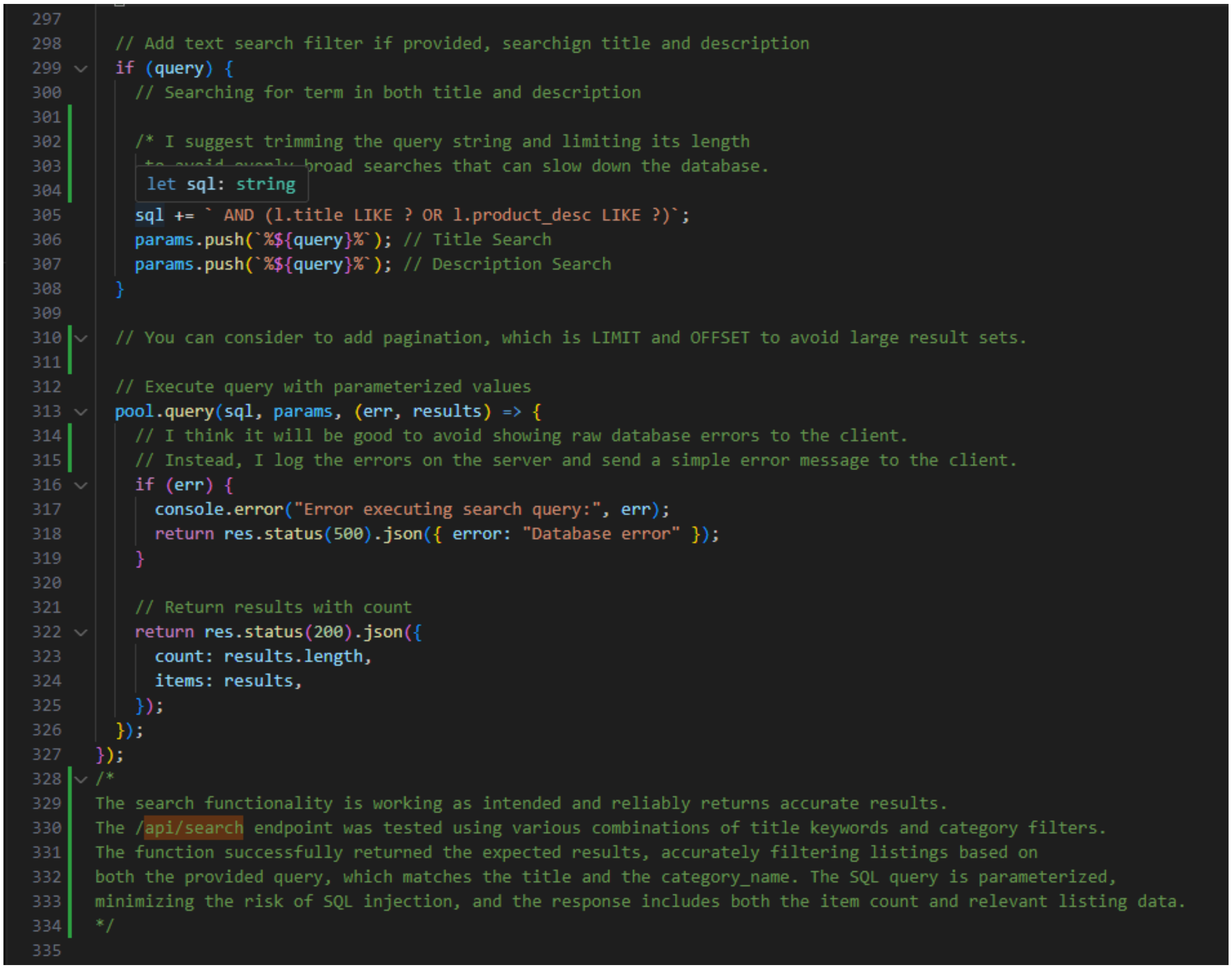
Please feel free to reach out if you need any further clarification or have additional questions.

Best regards,
Thanh Duong.

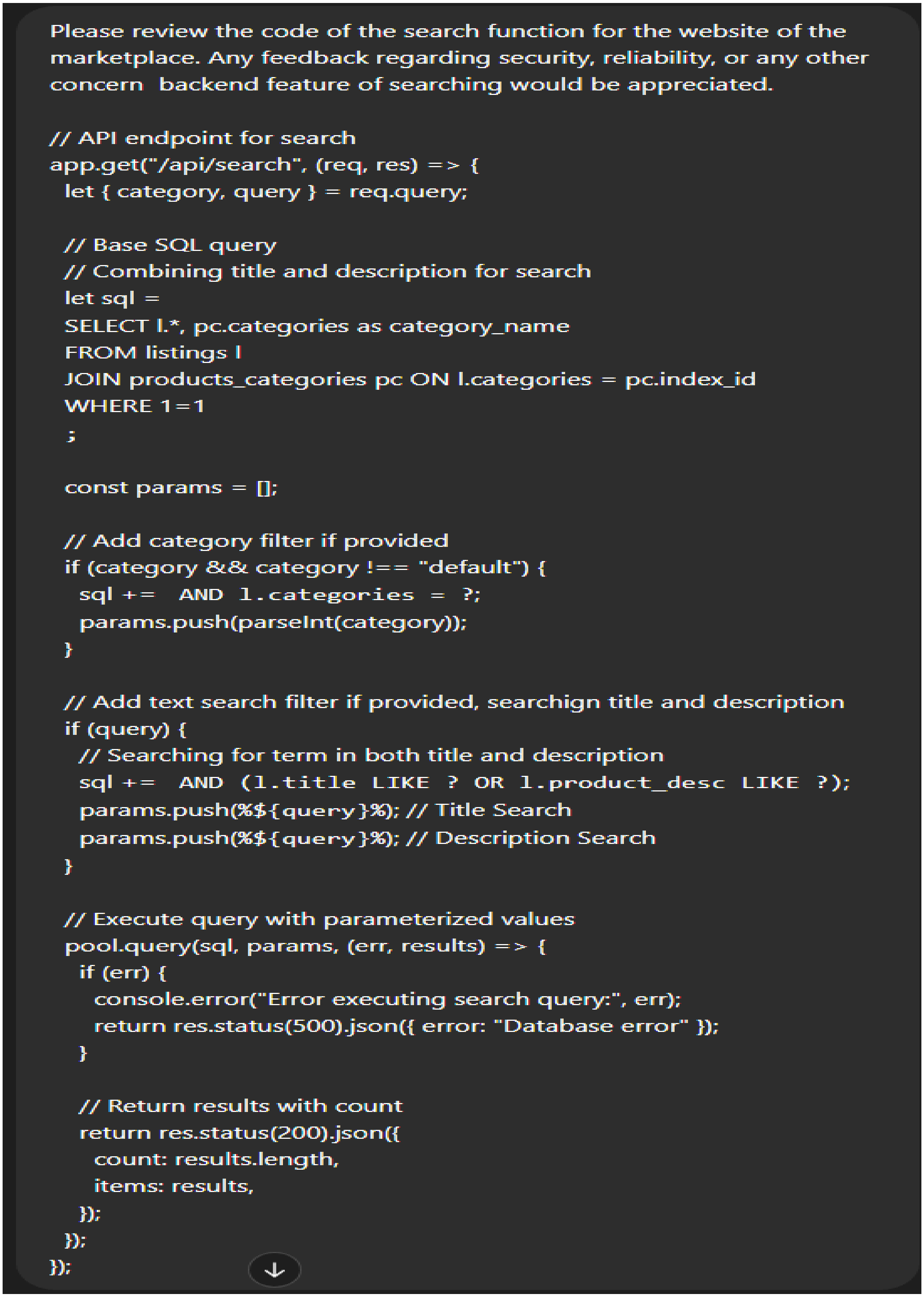
Code Comment Review



```
// API endpoint for search
app.get("/api/search", (req, res) => {
  // Extract category and query parameters from the request
  let { category, query } = req.query;

  // Base SQL query
  // Combining title and description for search
  let sql = `
  SELECT l.*, pc.categories as category_name
  FROM listings l
  JOIN products_categories pc ON l.categories = pc.index_id
  WHERE 1=1
  `;
  /*
  It is good to use  parameterized queries to prevent SQL injection.
  However, I recommend validating inputs like category and query to ensure
  they are the expected types and avoid errors or inefficient queries.
  */

  const params = [];

  // Add category filter if provided
  if (category && category !== "default") {
    // I would recommend validating that category is a valid integer before using it in the query.
    sql += ` AND l.categories = ?`;
    params.push(parseInt(category));
  }
```



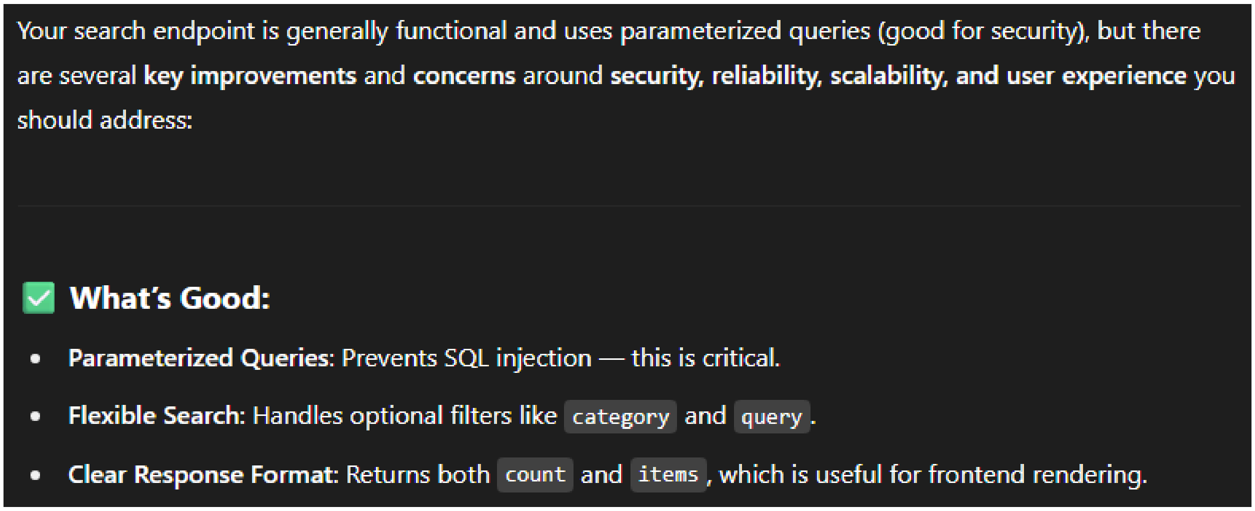
```
  // Add text search filter if provided, searchign title and description
  if (query) {
    // Searching for term in both title and description

    /* I suggest trimming the query string and limiting its length
    to avoid overly broad searches that can slow down the database.

    sql += ` AND (l.title LIKE ? OR l.product_desc LIKE ?)`;
    params.push(`%${query}%`); // Title Search
    params.push(`%${query}%`); // Description Search
  }

  // You can consider to add pagination, which is LIMIT and OFFSET to avoid large result sets.

  // Execute query with parameterized values
  pool.query(sql, params, (err, results) => {
    // I think it will be good to avoid showing raw database errors to the client.
    // Instead, I log the errors on the server and send a simple error message to the client.
    if (err) {
      console.error("Error executing search query:", err);
      return res.status(500).json({ error: "Database error" });
    }

    // Return results with count
    return res.status(200).json({
      count: results.length,
      items: results,
    });
  });
});
/*
The search functionality is working as intended and reliably returns accurate results.
The /api/search endpoint was tested using various combinations of title keywords and category filters.
The function successfully returned the expected results, accurately filtering listings based on
both the provided query, which matches the title and the category_name. The SQL query is parameterized,
minimizing the risk of SQL injection, and the response includes both the item count and relevant listing data.
*/

```



Please review the code of the search function for the website of the marketplace. Any feedback regarding security, reliability, or any other concern backend feature of searching would be appreciated.

```
// API endpoint for search
app.get("/api/search", (req, res) => {
  let { category, query } = req.query;

  // Base SQL query
  // Combining title and description for search
  let sql =
  SELECT l.*, pc.categories as category_name
  FROM listings l
  JOIN products_categories pc ON l.categories = pc.index_id
  WHERE 1=1
  ;

  const params = [];

  // Add category filter if provided
  if (category && category !== "default") {
    sql +=  AND l.categories = ?;
    params.push(parseInt(category));
  }

  // Add text search filter if provided, searchign title and description
  if (query) {
    // Searching for term in both title and description
    sql +=  AND (l.title LIKE ? OR l.product_desc LIKE ?);
    params.push(%${query}%); // Title Search
    params.push(%${query}%); // Description Search
  }

  // Execute query with parameterized values
  pool.query(sql, params, (err, results) => {
    if (err) {
      console.error("Error executing search query:", err);
      return res.status(500).json({ error: "Database error" });
    }

    // Return results with count
    return res.status(200).json({
      count: results.length,
      items: results,
    });
  });
});
```

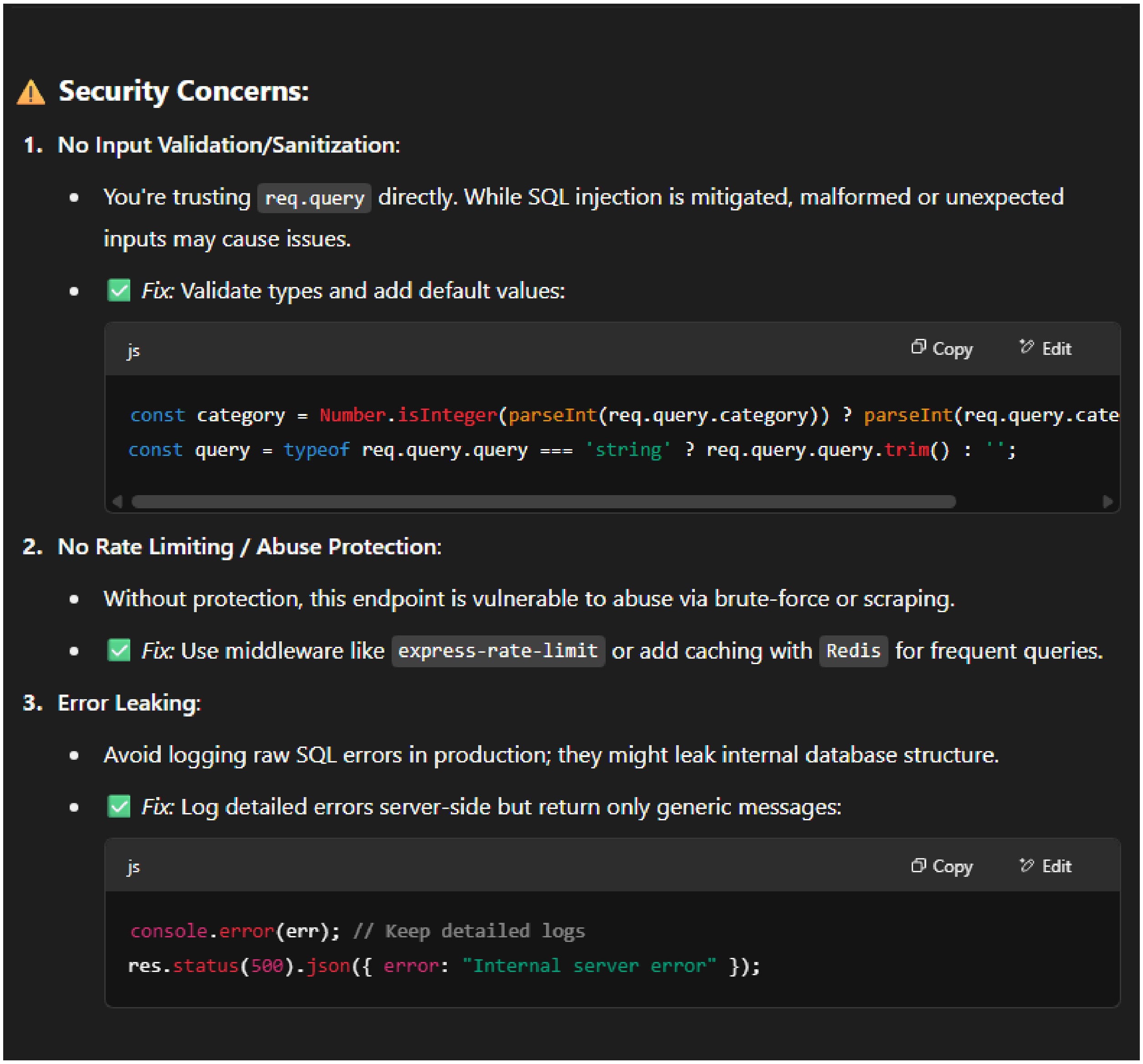
Output of GenAI for code review



Your search endpoint is generally functional and uses parameterized queries (good for security), but there are several **key improvements** and **concerns** around **security, reliability, scalability, and user experience** you should address:

✅ What's Good:

- **Parameterized Queries**: Prevents SQL injection — this is critical.
- **Flexible Search:** Handles optional filters like `category` and `query`.
- **Clear Response Format:** Returns both `count` and `items`, which is useful for frontend rendering.



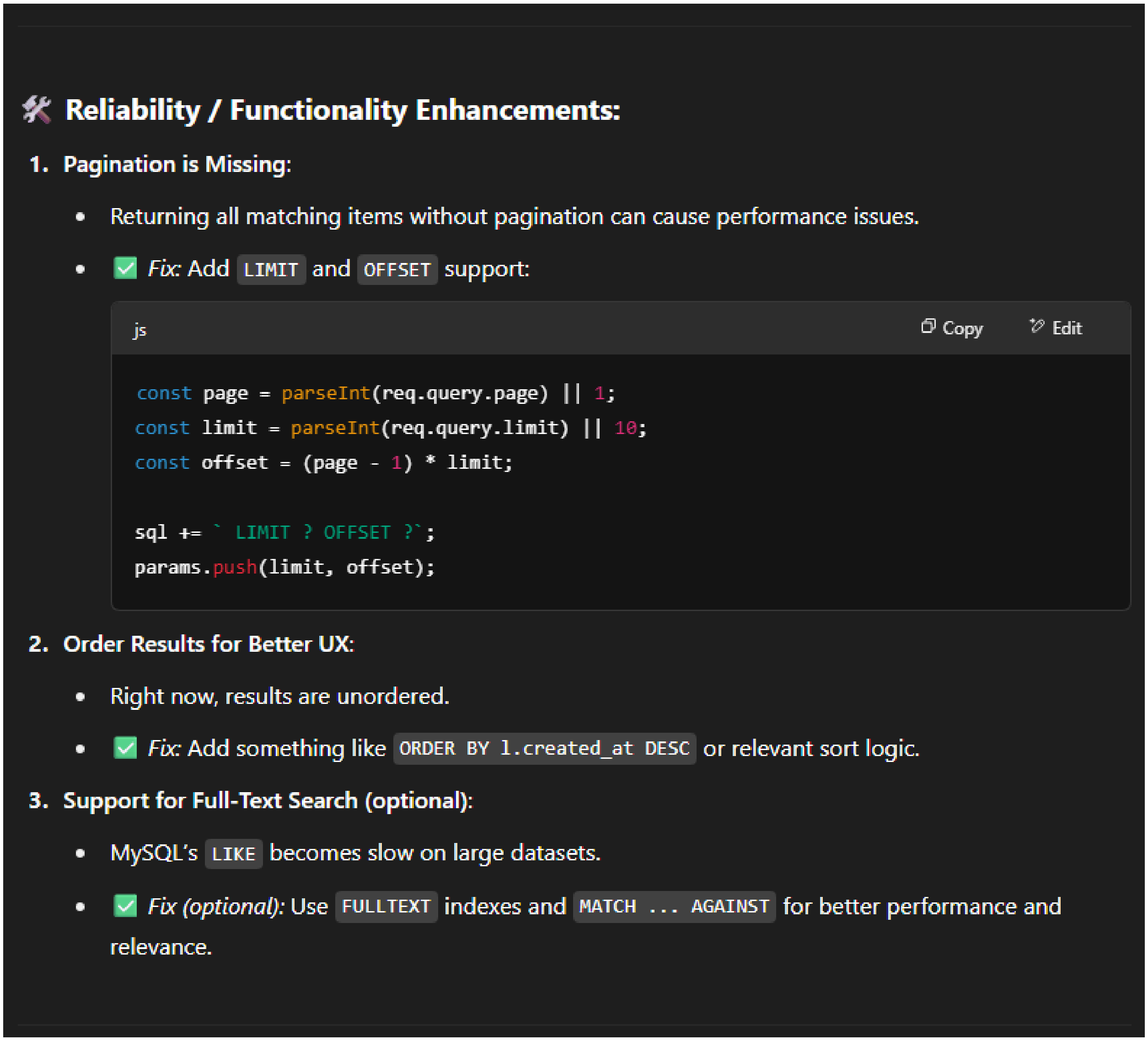
⚠️ Security Concerns:

1. **No Input Validation/Sanitization**:
    - You're trusting `req.query` directly. While SQL injection is mitigated, malformed or unexpected inputs may cause issues.
    - ✅ *Fix:* Validate types and add default values:

```js
const category = Number.isInteger(parseInt(req.query.category)) ? parseInt(req.query.cate
const query = typeof req.query.query === 'string' ? req.query.query.trim() : '';
```

2. **No Rate Limiting / Abuse Protection**:
    - Without protection, this endpoint is vulnerable to abuse via brute-force or scraping.
    - ✅ *Fix:* Use middleware like `express-rate-limit` or add caching with `Redis` for frequent queries.
3. **Error Leaking**:
    - Avoid logging raw SQL errors in production; they might leak internal database structure.
    - ✅ *Fix:* Log detailed errors server-side but return only generic messages:

```js
console.error(err); // Keep detailed logs
res.status(500).json({ error: "Internal server error" });
```

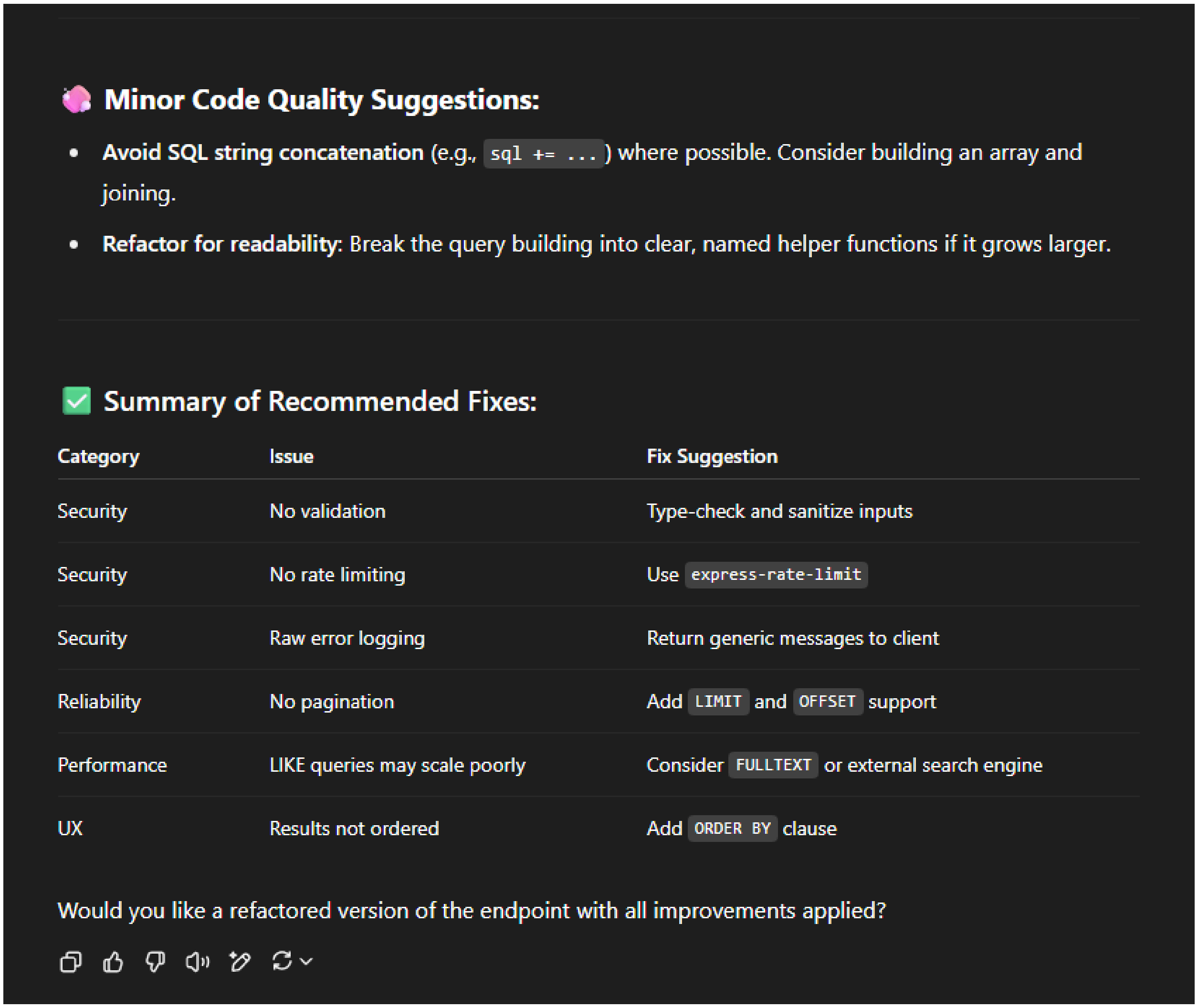


🛠️ Reliability / Functionality Enhancements:

1. **Pagination is Missing**:
   - Returning all matching items without pagination can cause performance issues.
   - ✅ *Fix:* Add `LIMIT` and `OFFSET` support:

```js
const page = parseInt(req.query.page) || 1;
const limit = parseInt(req.query.limit) || 10;
const offset = (page - 1) * limit;

sql += ` LIMIT ? OFFSET ?`;
params.push(limit, offset);
```

2. **Order Results for Better UX**:
   - Right now, results are unordered.
   - ✅ *Fix:* Add something like `ORDER BY l.created_at DESC` or relevant sort logic.
3. **Support for Full-Text Search (optional)**:
   - MySQL's `LIKE` becomes slow on large datasets.
   - ✅ *Fix (optional):* Use `FULLTEXT` indexes and `MATCH ... AGAINST` for better performance and relevance.



🧼 Minor Code Quality Suggestions:

- **Avoid SQL string concatenation** (e.g., `sql += ...`) where possible. Consider building an array and joining.
- **Refactor for readability**: Break the query building into clear, named helper functions if it grows larger.

✅ Summary of Recommended Fixes:

Category	Issue	Fix Suggestion
Security	No validation	Type-check and sanitize inputs
Security	No rate limiting	Use `express-rate-limit`
Security	Raw error logging	Return generic messages to client
Reliability	No pagination	Add `LIMIT` and `OFFSET` support
Performance	LIKE queries may scale poorly	Consider `FULLTEXT` or external search engine
UX	Results not ordered	Add `ORDER BY` clause

Would you like a refactored version of the endpoint with all improvements applied?

GenAI Usage

What genAI tool and version you used: The ChatGPT-4 model by OpenAI

Explain briefly how you used the tool and what benefit it offered:

The way I used the tool was that I told the AI about what I was looking for. I provided the code for the search function and asked for feedback. The tool offered valuable insights by reviewing the code for potential issues. It also suggested improvements to help me ensure the code is secure, efficient, and scalable.

Provide brief examples of key concepts and prompts:

- “Please review the code of the search function for the website of the marketplace. Any feedback regarding security, reliability, or any other concerns regarding the backend feature of searching would be appreciated.”
- “Can you help me ensure that my backend API doesn't expose raw database errors to the client in production?”
- “Can you review the way I handle user inputs in my backend search function, and suggest improvements for validating and sanitizing inputs?”
- “How can I optimize the performance of this search function if my database becomes larger and larger?”

Utility Ranking: Medium

BEST PRACTICES FOR SECURITY

This section outlines the major assets protected within GatorMarket, identifies potential security threats to each, and describes our strategies to mitigate those threats. We also confirm that best practices such as password encryption and input validation are implemented across the application.

Security Risk Table

Asset to be Protected	Types of Expected Attacks	Consequence of Security Breach	Strategy to Mitigate/Protect
User Passwords	Brute force, credential stuffing, SQL injection	Unauthorized access to user accounts	Passwords hashed using bcrypt before storage
User Email Addresses	Data scraping, phishing	Exposure of personal information, targeted attacks	Restricted email visibility; validation on input
Product Listings	Spam submissions, script injection (XSS)	Corruption of UI, loss of trust	Input validation and sanitization
User Sessions	Session hijacking, token theft	Impersonation of users	Session tokens with HTTP-only and secure flags
Messages	Spam or malicious links	User harassment, phishing	Only authenticated users can message; input validation

Search Input	Injection attacks, abuse	System crash, UI bugs	Limit input to 40 alphanumeric characters
Database	SQL injection	Data theft or loss	Use of parameterized queries and ORM protection

Confirmation of Security Practices

- Password encryption: All passwords are securely hashed using bcrypt before being stored in the database.

- Input Validation Enforced:

    - Search input: Limited to 40 alphanumeric characters

    - Registration email: Must end with ‘sfsu.edu’

    - Terms of Services: The checkbox must be accepted before registration can be completed

ADHERENCE TO ORIGINAL NON-FUNCTIONAL REQUIREMENTS

Non-functional Requirement	Status	Notes (if ISSUE)
Application shall be developed, tested and deployed using tools and cloud servers approved by Class CTO and as-agreed in M0	DONE	
Application shall be optimized for standard desktop/laptop browsers e.g. must render correctly on the two latest version of two major browsers	DONE	
All or selected application functions shall render well on mobile devices (no native app to be developed)	DONE	
Posting of sales information and messaging to sellers shall be limited only to SFSU students	DONE	

Critical data shall be stored in the database on the team’s deployment server	DONE	
No more than 50 concurrent users shall be accessing the application at any time	ON TRACK	System supports 10+ users; more concurrency testing pending
Privacy of users shall be protected	DONE	Data is encrypted and access-controlled
The language used shall be English (no localization needed)	DONE	
Application shall be very easy to use and intuitive	ON TRACK	Usability testing in progress
Application shall follow established architecture patterns	DONE	
Application code and its repository shall be easy to inspect and maintain	ON TRACK	Code is structured, commented, and in GitHub
Google Analytics shall be used	ISSUE	Not implemented yet
No e-mail client or chat services shall be allowed. Messaging is limited to one-way in-site messages	ON TRACK	Only seller messaging is implemented
Pay functionality, if any, shall not be implemented nor simulated in UI	DONE	
Site security: basic best practices shall be applied	DONE	See Section 5 for details
Media formats shall be standard as used in the market today	DONE	Image upload supports JPG, PNG
Modern SE processes and tools shall be used	DONE	Using GitHub, issue tracking, and VSCode
Application UI shall display: 'SFSU Software Engineering Project CSC 648-848, Spring 2025. For Demonstration Only' at the top of each page	ON TRACK	Banner added to some pages; rollout to all pages in progress