SW Engineering CSC648-848

Fall 2025

Team 08

Kojiro Miura Kmiura@sfsu.edu	**Team Lead**
Atharva Walawalkar	**Backend Lead**
Addhyan Kohli	**Frontend Lead**
Krinjal Basnet	**Frontend Dev**
Aketzali Zeledon	**Backend Dev**
Sonam Tobgyal	**Github Master**

URL: http://3.101.155.82/

Initial Submission	12/16/2025
Revised submission	N/A

Table of Contents

1. Product Summary
2. Usability test plan
3. QA test plan and QA testing
4. Peer Code Review
5. Self-check on best practices for security
6. Self-check of the adherence to original Non-functional specs
7. Use of GenAI tools like ChatGPT To create a Test Suite for Registration

Product Summary

Name of product: Gator Tutor

Product Description:

Gator Tutor is an exclusive academic platform connecting San Francisco State University students with trusted and verified tutors in a secure environment. The platform allows students to search for tutors based on courses, filter by various criteria, and schedule tutoring sessions. Tutors can manage their profiles, set availability, and approve/reject booking requests. The system maintains academic credibility by requiring SFSU email addresses and admin approval of tutor profiles.

What is unique in your product:

1. **SFSU-exclusive access** - Only SFSU community members can register and use the platform
2. **Advanced tutor search** - Filtering with multiple criteria
3. **Integrated scheduling system** - Complete booking workflow with tutor approval
4. **Course management** - Tutors can add/remove courses they teach, with admin approval
5. **Comprehensive reporting system** - Users can report issues, admins can manage reports
6. **Real-time chat system** - In-app messaging with file/media sharing capabilities
7. **Role-based access control** - Distinct workflows for students, tutors, and admins

Final Committed P1 Functions:

1. Users can register with SFSU email addresses
2. Users can log in and log out securely
3. Unregistered users can browse and search tutor profiles
4. Users can search tutors with advanced filters
5. Users can view detailed tutor profiles with courses and availability
6. Students can book tutoring sessions with specific date/time
7. Tutors can approve or reject booking requests
8. Tutors can manage their recurring weekly availability schedule
9. Tutors can update their profile information

10. Tutors can add/remove courses they teach
11. Users can search for courses and see available tutors
12. System provides filter options for search UI
13. Admin can approve/reject tutor profiles
14. Admin can manage course listings (add, deactivate, update)
15. Admin can manage tutor course requests (approve/reject)
16. Users can submit reports on other users
17. Admin can view and manage all user reports
18. Real-time chat messaging between users
19. File/media sharing in chat
20. Booking system prevents double-booking
21. Availability slots expire automatically based on semester duration

Usability test plan

Tutor Search Function

1. Test Objectives

Test the tutor search functionality to ensure users can effectively find tutors using various filters. Evaluate the search interface for intuitiveness, responsiveness, and effectiveness in helping users find relevant tutors based on multiple criteria including price, availability, languages, and courses.

2. Test Background and Setup

System Setup:

- Backend: FastAPI server running on Ubuntu with MySQL database
- Frontend: React application (to be tested)
- Database: Contains sample tutor data with various attributes (courses, rates, languages, availability)

- Backend Modules: Authentication, Registration, Admin Panel, Chat System, Database Management

Starting Point:

Tester begins at the main search page with access to search bar and filter panel.

Hardware Requirements:

- Laptop or desktop computer
- Stable internet connection
- Modern web browser (Chrome/Firefox latest versions)

Intended Users:

- SFSU students looking for tutors
- Users with varying technical proficiency

3. Usability Task Description

Instructions to Tester:
"You are an SFSU student looking for a tutor for your CSC 210 course. Please use the tutoring portal to find a suitable tutor by following these steps:

1. Navigate to the tutor search page
2. Search for tutors who teach CSC courses
3. Filter the results to show only tutors available on Mondays
4. Further filter to show only tutors charging less than $40 per hour
5. Find a tutor who speaks English
6. Book a session with a selected tutor for next Monday at 10:00 AM
7. Return to search results and sort them by price (lowest first)
8. Try searching for a specific tutor by name

Please think aloud as you perform these tasks, explaining what you're trying to do and any difficulties you encounter."

4. Plan for Evaluation of Effectiveness

Effectiveness will be measured by task completion rate. For each of the 8 tasks above, we will record whether the user successfully completes it. Success criteria: User achieves the intended outcome within 3 minutes per task without assistance. Effectiveness score = (Completed tasks / Total tasks) × 100.

5. Plan for Evaluation of Efficiency

Efficiency will be measured by time-on-task for each completed task. We will record the time taken from task instruction to successful completion. We'll calculate average time per task and compare against benchmark times (30 seconds for simple tasks like searching, 2 minutes for complex tasks like booking). Efficiency score = (Benchmark time / Actual time) × 100 for each task.

6. Plan for Evaluation of User Satisfaction (Likert Scale Questionnaire)

After completing the tasks, please rate your agreement with the following statements:

1. The search function was easy to use.

- Strongly Disagree
- Disagree
- Neutral
- Agree
- Strongly Agree

2. The filter options helped me find relevant tutors quickly.

- Strongly Disagree
- Disagree
- Neutral
- Agree
- Strongly Agree

3. I was able to complete my search tasks without frustration.

- Strongly Disagree
- Disagree
- Neutral
- Agree
- Strongly Agree

4. The search results were relevant to my criteria.

- Strongly Disagree
- Disagree
- Neutral
- Agree
- Strongly Agree

5. I would use this search function regularly to find tutors.

- Strongly Disagree
- Disagree
- Neutral
- Agree
- Strongly Agree

QA test plan and QA testing

1. Test Objectives

Test the tutor search API endpoints for correct functionality, error handling, and performance with various filter combinations. Test all backend modules: Authentication, Registration, Admin Panel, Chat System.

2. HW and SW Setup

- Hardware: Test server with Ubuntu, 2GB RAM

- Software: FastAPI 0.116.2, MySQL 8.0.43, Python 3.11, SQLAlchemy 2.0.44
- Test tools: curl, Postman, custom bash scripts, pytest
- URL: http://127.0.0.1:8000 (local testing)

3. Feature to be Tested

Complete backend API system including:

- Authentication endpoints (login, user management)
- Registration system
- Admin endpoints (tutor management, course management, reports)
- Chat system (text and media messages, WebSocket)
- Tutor search and filtering

4. QA Test Plan Table

Test #	Test Title	Test Description	Test Input	Expected Correct Output	Test Results
1	User Login	Test valid user login	POST /api/login with valid credentials	200 OK with token and user_id	
2	Invalid Login	Test invalid credentials	POST /api/login with wrong password	401 Unauthorized	

3	Get User Info	Get user by ID	GET /api/users/{user_id}	200 OK with user details
4	Get All Users	Admin view all users	GET /api/users	200 OK with user list
5	User Registration	Register new user	POST /api/register with SFSU email	201 Created with user_id
6	Duplicate Email	Register with existing email	POST /api/register with existing email	400 Bad Request
7	Non-SFSU Email	Register with non-SFSU email	POST /api/register with gmail.com	400 Bad Request
Admin Module - Tutor Management				
8	Student Submit Tutor Application	Submit Application from student to Admin	POST /api/admin/tutor-appicatio ns	200 OK with confirmation of rejection or approval

9	Update Application Status	Reject tutor application, OR Approve and promote student to Tutor	PATCH /api/admin/tutor-application/{application_id}/status	200 OK with confirmation of updated status
10	Get Tutor Applications	View tutor applications	GET /api/admin/all-tutor-applications	200 OK with applications list
Admin Module - Course Management				
11	Get All Courses	View all courses	GET /api/admin/allcourses	200 OK with course list
12	Get All Course Coverage Requests	Admin View all Course Coverage Requests	GET /api/admin/all-coverage-requests	200 OK with course requests list
13	Submit Course	Request new course.	GET /api/admin/submit-coverage-request	200 OK with confirmation

	Coverage Request			
14	Update Course Request	Reject Request OR Approve and add course.	PATCH /api/admin/coverage-request/{request_id}/status	200 OK with updated request status
Admin Module - Tutor Course Requests				
15	Get Course Requests	View all tutor course requests	GET /api/admin/all-tutor-course-requests	200 OK with requests list
16	Create Course Request	Tutor requests new course	POST /api/admin/tutor-course-request/{tutor_id}	200 OK with request details
17	Approve Course Request	Admin approves request	PATCH /api/admin/tutor-course-request/{request_id}/approve	200 OK with updated request

18	Reject Course Request	Admin rejects request	PATCH /api/admin/tutor-course-request/{request_id}/reject	200 OK with updated request
19	Remove Tutor Course	Admin removes tutor course	DELETE /api/admin/tutor/{tutor_id}/course/{course_id}	200 OK with confirmation
Admin Module - Reports				
20	Create Report	User submits report	POST /api/admin/report	201 Created with report details
21	Get All Reports	Admin views all reports	GET /api/admin/allreports	200 OK with reports list
22	Update Report Status	Admin updates report status	PATCH /api/admin/report/{report_id}/status	200 OK with updated report
Chat Module				

23	Send Text Message	Send text message	POST /api/chat/send with text content	200 OK with message details
24	Send Media Message	Send message with file	POST /api/chat/send-media with file	200 OK with message details
25	Get Chat History	View chat between users	GET /api/chat/chatroomhistory/{user1}/{user2}	200 OK with messages list
26	Get User Chats	View all user's chats	GET /api/chat/allchats/{user_id}	200 OK with user IDs list
27	WebSocket Connection	Real-time messaging	WS /api/chat/ws/{user_id}	Successful connection
Search Module				
28	Basic Search	Search tutors	GET /search/tutors	200 OK with tutor list

29	Filter by Department	Filter tutors by department	GET /search/tutors?department=CSC	200 OK, tutors in CSC department
30	Price Range Filter	Filter by price range	GET /search/tutors?min_rate=2000&max_rate=4000	200 OK, tutors in price range
31	Language Filter	Filter by language	GET /search/tutors?languages=Spanish	200 OK, tutors speaking Spanish
Error Cases				
32	Invalid User ID	Access non-existent user	GET /api/users/99999	404 Not Found
33	Invalid Course ID	Access non-existent course	GET /api/admin/courses/99999	404 Not Found
34	Unauthorized Access	Access admin	POST /api/admin/addcourse without admin role	403 Forbidden

without
permission

GenAI Use in QA Test Plan

GenAI Tool Used: ChatGPT-4, version as of December 2025

How GenAI was used:
I used ChatGPT to:

1. Generate comprehensive test cases covering edge cases
2. Suggest test input variations
3. Help structure the test plan table
4. Identify potential error conditions to test
5. Generate test data for database population

Key Examples and Prompts:

- "Generate test cases for a complete tutoring platform backend with authentication, admin panel, chat, and search functionality"
- "What edge cases should I test for a report submission system?"
- "Create test data for tutor profiles with various attributes"
- "Generate SQL injection test cases for search endpoints"

Benefit Offered:
GenAI helped identify test cases I might have missed, particularly edge cases and error conditions. It suggested a logical progression of tests from simple to complex and helped create comprehensive test coverage across all modules.

Rank Utility of GenAI: HIGH

Peer Code Review

Need to do still

Self-check on best practices for security

Asset to be Protected	Types of Possible/Expected Attacks	Consequence of Security Breach	Your Strategy to Mitigate/Protect the Asset
User Credentials	Brute force attacks, credential stuffing	Account takeover, unauthorized access	1. Password hashing 2. Rate limiting on login attempts 3. SFSU email validation 4. Session timeout implementation
User Data (PII)	Data scraping, SQL injection	Privacy violation, identity theft	1. SQLAlchemy ORM prevents SQL injection 2. Input validation on all endpoints 3. Role-based access control 4. Data encryption at rest (planned)

Tutor Profiles	Fake profiles, misinformation	Platform credibility loss, fraud	1. Admin approval required for tutor profiles 2. SFSU email verification mandatory 3. Report and flag system for inappropriate content 4. Regular profile audits
Booking System	Double-booking attacks, schedule manipulation	Service disruption, user frustration	1. Transaction locking for concurrent bookings 2. Validation of time slot availability 3. Tutor approval workflow 4. Audit logs for all booking changes
Database	SQL injection, unauthorized access	Data theft, data corruption	1. Parameterized queries via SQLAlchemy 2. Limited database user permissions 3. Regular backups 4. Database firewall rules

Chat Messages	Eavesdropping, message tampering	Privacy violation, harassment	1. WebSocket with authentication 2. Message encryption (planned) 3. Report inappropriate messages 4. Message retention policies
File Uploads	Malware upload, phishing attacks	System compromise, malware spread	1. File type validation (images, PDFs only) 2. Size limits (10MB max) 3. Virus scanning (planned) 4. Secure file storage with access controls
Admin Functions	Privilege escalation, unauthorized changes	System-wide compromise	1. Strict role-based access control 2. Admin activity logging 3. Two-factor authentication (planned) 4. Regular security audits

Search Functionality	DoS via complex searches, data scraping	Service downtime, data theft	1. Query timeout limits (5 seconds) 2. Rate limiting per IP 3. Search complexity limits 4. CAPTCHA for suspicious activity

Password Encryption: CONFIRMED - Passwords are hashed using Argon2 algorithm in both auth and registration modules.

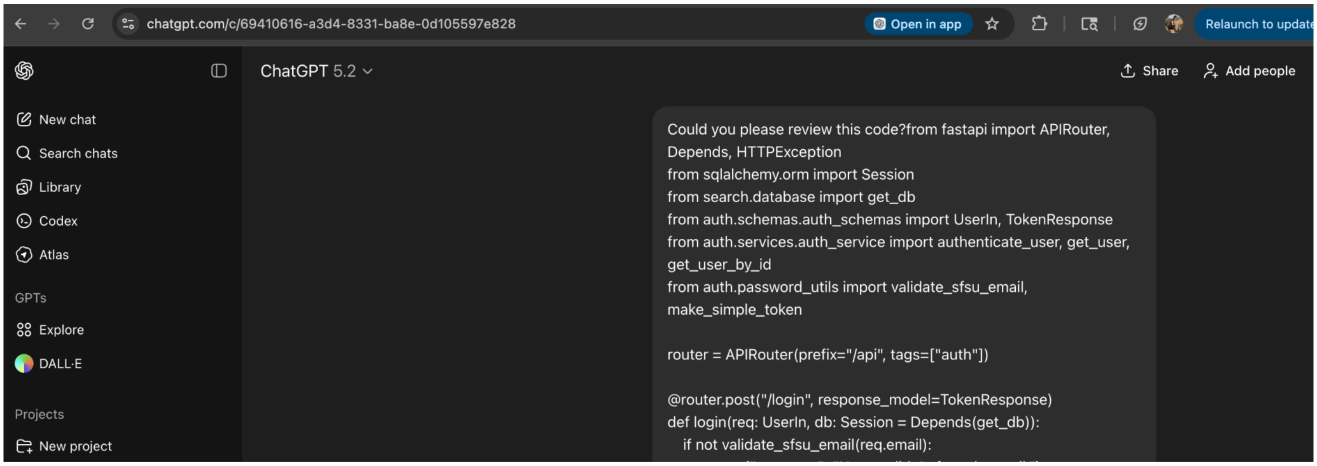
Input Data Validation: IMPLEMENTED

1. **Search bars**: Limited to 100 characters (validated in router parameters)
2. **SFSU email validation**: Must include "@sfsu.edu" (implemented in auth/password_utils.py)
3. **Terms acceptance**: Required in registration form (frontend implementation)
4. **Additional validations**:
    - Price must be positive integers
    - Date formats validated with Pydantic
    - Enum values checked for status fields
    - File type validation for uploads
    - Length limits on text fields

Self-check of the adherence to original Non-functional specs

1. Application shall be developed, tested and deployed using tools and servers approved by Class CTO - **ON TRACK**
2. Application shall be optimized for standard desktop/laptop browsers - **ON TRACK**
3. All or selected application functions shall be rendered well on mobile devices - **ON TRACK**
4. Posting of tutor information and messaging to tutors shall be limited only to SFSU students - **DONE**
5. Critical data shall be stored in the database on the team's deployment server - **DONE**
6. No more than 50 concurrent users shall be accessing the application at any time - **ON TRACK**
7. Privacy of users shall be protected - **ON TRACK**
8. The language used shall be English - **DONE**
9. Application shall be very easy to use and intuitive - **ON TRACK**
10. Application shall follow established architecture patterns - **DONE**
11. Application code and its repository shall be easy to inspect and maintain - **DONE**
12. Google analytics shall be used - **DONE**
13. No e-mail clients shall be allowed. Interested users can only message via in-site messaging - **DONE**
14. Pay functionality shall not be implemented nor simulated in UI - **DONE**
15. Site security: basic best practices shall be applied - **ON TRACK**
16. Media formats shall be standard as used in the market today - **ON TRACK**
17. Modern SE processes and tools shall be used - **DONE**
18. The application UI shall prominently display required text - **ON TRACK** (Frontend will display "SFSU Software Engineering Project CSC 648-848, Fall 2025. For Demonstration Only")

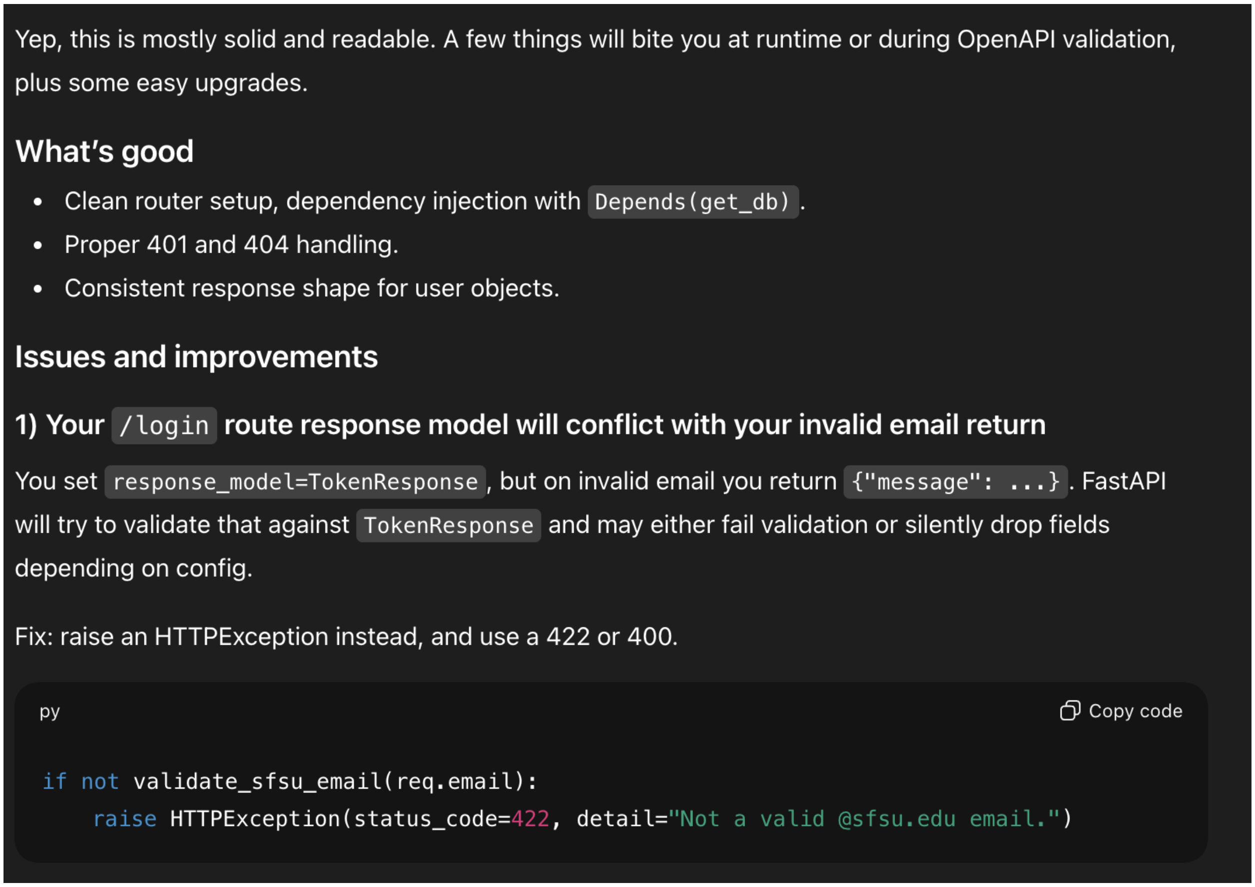
Use of GenAI tools like ChatGPT To create a Test Suite for Registration



Could you please review this code?from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from search.database import get_db
from auth.schemas.auth_schemas import UserIn, TokenResponse
from auth.services.auth_service import authenticate_user, get_user, get_user_by_id
from auth.password_utils import validate_sfsu_email, make_simple_token

router = APIRouter(prefix="/api", tags=["auth"])

@router.post("/login", response_model=TokenResponse)
def login(req: UserIn, db: Session = Depends(get_db)):
  if not validate_sfsu_email(req.email):



Yep, this is mostly solid and readable. A few things will bite you at runtime or during OpenAPI validation, plus some easy upgrades.

What’s good

- Clean router setup, dependency injection with `Depends(get_db)`.
- Proper 401 and 404 handling.
- Consistent response shape for user objects.

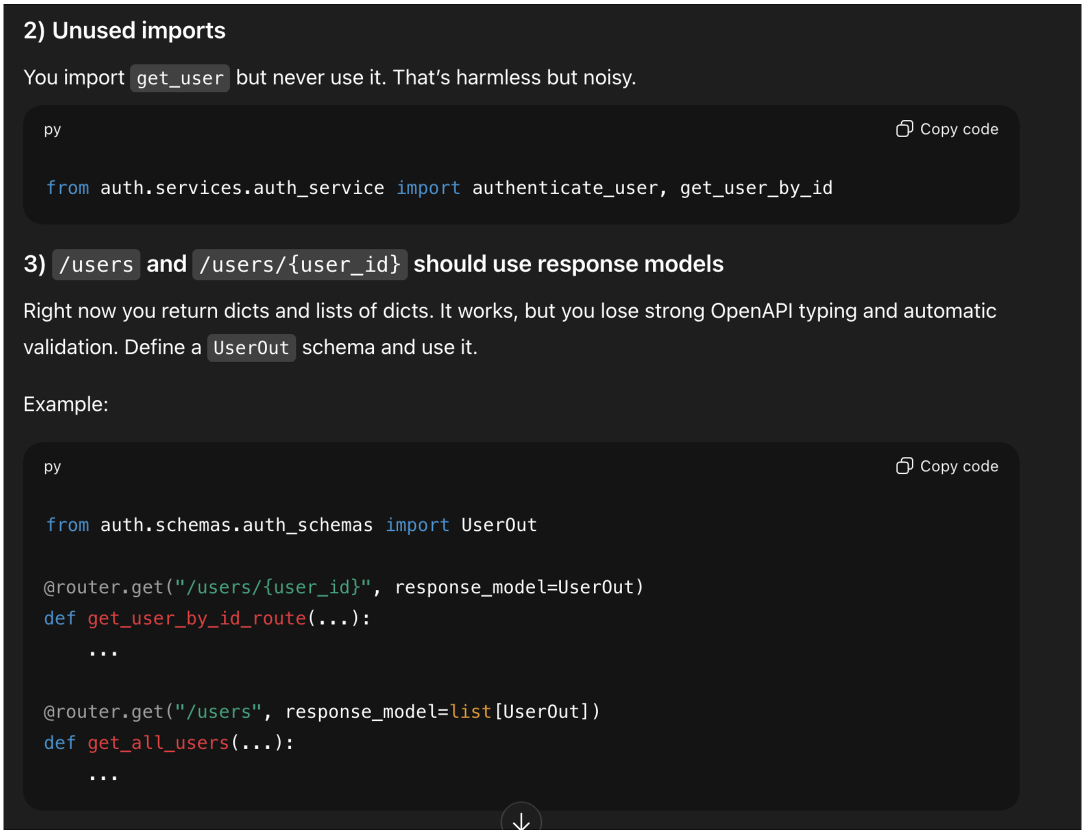
Issues and improvements

1) Your `/login` route response model will conflict with your invalid email return

You set `response_model=TokenResponse`, but on invalid email you return `{"message": ...}`. FastAPI will try to validate that against `TokenResponse` and may either fail validation or silently drop fields depending on config.

Fix: raise an HTTPException instead, and use a 422 or 400.

```py
if not validate_sfsu_email(req.email):
    raise HTTPException(status_code=422, detail="Not a valid @sfsu.edu email.")
```



2) Unused imports

You import `get_user` but never use it. That's harmless but noisy.

```py
from auth.services.auth_service import authenticate_user, get_user_by_id
```

3) `/users` and `/users/{user_id}` should use response models

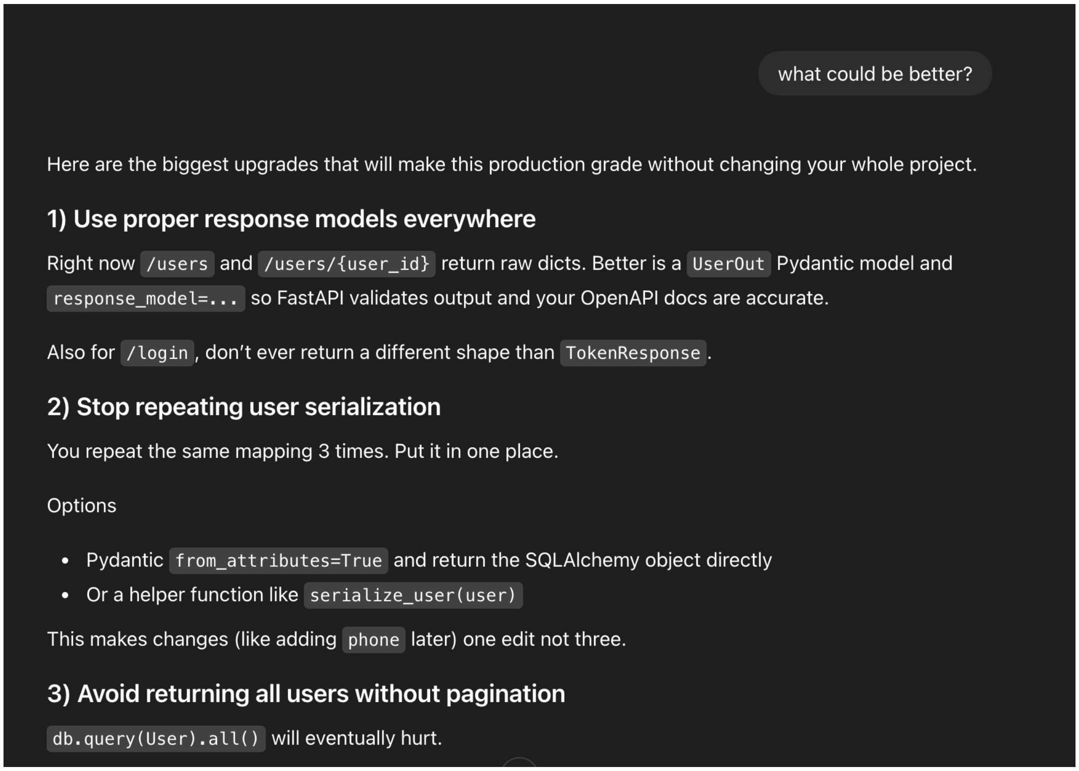
Right now you return dicts and lists of dicts. It works, but you lose strong OpenAPI typing and automatic validation. Define a `UserOut` schema and use it.

Example:

```py
from auth.schemas.auth_schemas import UserOut

@router.get("/users/{user_id}", response_model=UserOut)
def get_user_by_id_route(...):
    ...

@router.get("/users", response_model=list[UserOut])
def get_all_users(...):
    ...
```



what could be better?

Here are the biggest upgrades that will make this production grade without changing your whole project.

1) Use proper response models everywhere

Right now `/users` and `/users/{user_id}` return raw dicts. Better is a `UserOut` Pydantic model and `response_model=...` so FastAPI validates output and your OpenAPI docs are accurate.

Also for `/login`, don't ever return a different shape than `TokenResponse`.

2) Stop repeating user serialization

You repeat the same mapping 3 times. Put it in one place.

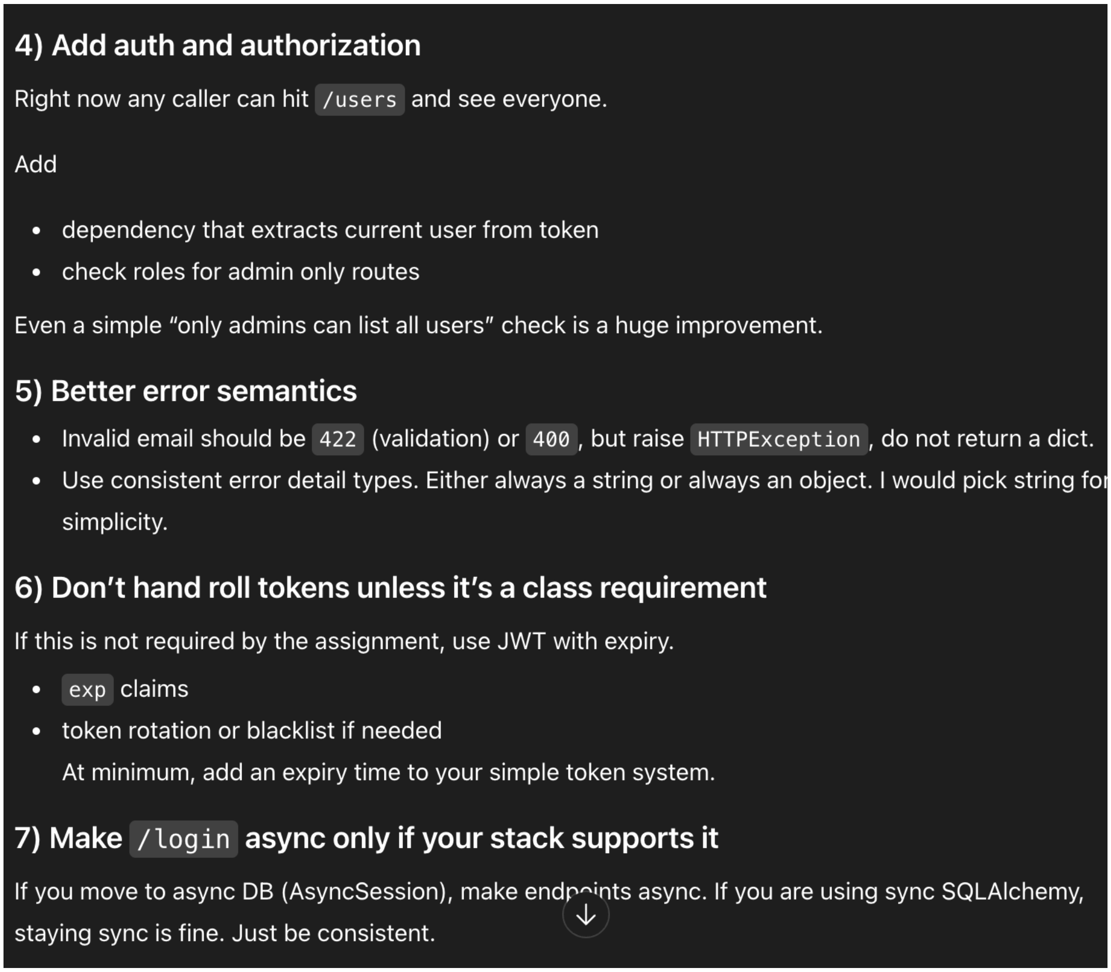
Options

- Pydantic `from_attributes=True` and return the SQLAlchemy object directly
- Or a helper function like `serialize_user(user)`

This makes changes (like adding `phone` later) one edit not three.

3) Avoid returning all users without pagination

`db.query(User).all()` will eventually hurt.



4) Add auth and authorization

Right now any caller can hit `/users` and see everyone.

Add

- dependency that extracts current user from token
- check roles for admin only routes

Even a simple "only admins can list all users" check is a huge improvement.

5) Better error semantics

- Invalid email should be `422` (validation) or `400`, but raise `HTTPException`, do not return a dict.
- Use consistent error detail types. Either always a string or always an object. I would pick string fo simplicity.

6) Don't hand roll tokens unless it's a class requirement

If this is not required by the assignment, use JWT with expiry.

- `exp` claims
- token rotation or blacklist if needed
  At minimum, add an expiry time to your simple token system.

7) Make `/login` async only if your stack supports it

If you move to async DB (AsyncSession), make endpoints async. If you are using sync SQLAlchemy, staying sync is fine. Just be consistent.

Tool: ChatGPT GPT 5.2
Prompt Example
Review this auth_router.py FastAPI router for readability, consistency with FastAPI best practices, potential bugs, security risks, and API design issues. What could be improved and why?

Brief Explanation of Tool Use and Benefits
I used GenAI as a coding assistant to review and refine the authentication router. The tool inspected the login and user retrieval endpoints, then highlighted issues related to response consistency, validation, and security that are easy to miss during normal development. It also suggested best practice improvements for API design, including stronger typing with Pydantic response models, cleaner error handling, and safer access control for user listing endpoints. The main benefit was speed and coverage. Instead of manually hunting for edge cases, I received an organized list of concrete improvements with clear reasoning and examples, which helped me prioritize fixes and improve maintainability.

AI Findings

Strengths

Clear router structure and route naming
Good use of Depends(get_db) for database session injection
Appropriate use of HTTPException for 401 and 404 scenarios
Readable logic flow and consistent user field mapping

Improvements

API correctness and consistency
If response_model is TokenResponse, the endpoint should not return a different shape on invalid email. Use HTTPException (422 or 400) so output always matches the response model
Use response_model for /users and /users user_id to enforce consistent output and improve OpenAPI documentation

Maintainability

Remove unused imports to keep the module clean
Avoid repeating the same user dict mapping by using a shared serializer or a UserOut Pydantic model with from_attributes enabled
Move model imports to the top of the file unless you are intentionally avoiding circular imports

Security and access control

Protect GET /users to prevent exposing all user records to unauthenticated callers
Add role based authorization for admin only routes
Add rate limiting or logging for repeated failed login attempts to reduce brute force risk
Use tokens with expiry rather than long lived simple tokens, unless the project requirements specify otherwise

Performance and scalability

Add pagination parameters (skip, limit) for GET /users instead of loading all users at once

Utility Ranking: HIGH

AI caught subtle improvement opportunities not easily noticed.