Team S

LLMscribe
Software Requirements Specification
For an LLM-based cooperative editor

Version <1.0>

Revision History

Date	Version	Description	Author
03/18/2025	1.0	1st Phase Report of LLM-based cooperative editor project	Sean Jenkins, Adama Faye, Rajiv Seeram, Nischal Thapa, Junaet Mahbub

Table of Contents

Software Requirements Specification

1. Introduction

1.1 Purpose

The purpose of this document is to explain what the LLM Editor should do. The software will:

- Let users write or paste text and get OpenAI's GPT models suggestions for improvement.
- Use OpenAI's GPT models to help with writing, editing, and summarization.
- Provide a simple and user-friendly text editor.
- Use Flask as the backend to process user input and send AI responses.
- The frontend will be made using Flask's Jinja templating, with JavaScript, Python, and TailwindCSS
- The purpose of this use of technologies is to find a secure system that seamlessly integrates with the ability to expand further - Flask is the perfect gateway to full stack development.

1.2 Scope

The LLM Editor will act as an AI-powered writing assistant, allowing users to improve their text using AI models. It will offer:

- Key Features
- Text Editing and Formatting: Users can type, edit, and format text like a traditional editor.
- AI-Powered Suggestions: AI will provide grammar fixes, rewording, and style improvements.
- Summarization and Paraphrasing: Users can summarize long text or rephrase sentences for better readability.
- Content Generation: AI can generate text based on a given prompt.
- Real-Time AI Processing: AI responses will be generated instantly using the OpenAI API.
- User-Friendly Interface: The app will use Next.js for smooth and fast navigation.
- Backend: Flask will handle all API requests, ensuring security and scalability.

Technology Stack

- Frontend: Flask- Jinja
- Backend: Flask (Python-based server for processing requests)
- AI Integration: OpenAI API (for AI-powered editing and writing)
- Database: AWS

The system will be designed to run efficiently on browsers.

1.3 Definitions, Acronyms, and Abbreviations

Flask allows you to create dynamic web applications by handling HTTP requests, rendering HTML templates, and interacting with databases.

This largely depends on the file structure- here in this prototype is the file structure of the current project as shown in Appendix A.

In Flask, the application architecture is structured around **models**, **database schemas**, and **routes**. The models define the data structures that the database will translate into schemas, while the routes handle the various functions sent by the https requests from the client side.

ORM (Object-Relational Mapping): A technique that allows developers to interact with a database using Python objects instead of writing raw SQL queries. In this project, Flask-SQLAlchemy is used as the ORM.

RDS (Relational Database Service): A managed database service provided by AWS (Amazon Web Services). In this project, AWS RDS MySQL is used as the database.

Session Management: A mechanism for maintaining user state across multiple HTTP requests. Flask uses sessions to store user-specific data (e.g., user ID) after login. This is what we will use to time the

Template Rendering: The process of generating HTML dynamically using placeholders and logic. Flask uses the Jinja2 templating engine to render HTML templates.

Static Files: Files like CSS, JavaScript, and images that are served directly to the client without modification. In Flask, these are stored in the static/ folder.

Blueprint: A way to organize Flask applications into reusable components. In this project, the LLM module uses a Blueprint to define routes.

1.4 References

1. Flask Documentation
    - Title: Flask Documentation
    - Source: Flask Official Website
    - Date: Latest version as of October 2023
    - Availability: https://flask.palletsprojects.com/
2. Flask-SQLAlchemy Documentation
    - Title: Flask-SQLAlchemy Documentation
    - Source: Flask-SQLAlchemy Official Website
    - Date: Latest version as of October 2023
    - Availability: https://flask-sqlalchemy.palletsprojects.com/
3. AWS RDS Documentation
    - Title: Amazon RDS Documentation
    - Source: Amazon Web Services (AWS)
    - Date: Latest version as of October 2023
    - Availability: https://aws.amazon.com/rds/
4. MySQL Documentation
    - Title: MySQL Documentation
    - Source: MySQL Official Website
    - Date: Latest version as of October 2023
    - Availability: https://dev.mysql.com/doc/

5. Jinja2 Documentation
    - Title: Jinja2 Documentation
    - Source: Jinja2 Official Website
    - Date: Latest version as of October 2023
    - Availability: https://jinja.palletsprojects.com/

1.5 Overview

This document outlines the functional and operational requirements of our application. Section 2 provides a high-level overview of the system, defining the three user roles—Free Users, Paid Users, and Super Users—and detailing the core functionalities and governing rules. Section 3 presents comprehensive use cases, including user authentication, text submission and correction, token management, blacklist updates, and complaint resolution.

2. Overall Description

2.1 Use-Case Model Survey

Free users use case:

- Maximum of 20 words to submit to the text editor, and if exceeded the user will be thrown out of the product services.
- When thrown out, the user will no longer be able to log in for a duration of three minutes.

Paid users use case:

- Submit text with a token based restriction. Tokens can be acquired to increase the amount of words the user wants to use.
- Granted a feature for self correction to manually correct their text or LLM correction to improve output before submission. Then capable of saving those changes.
- Ability to report disputes.

Super users use case:

- Manage sign-ups, complaints, and penalties for both free and paid users as needed.
- Handle blacklist word submissions so that the system automatically blocks content related to these words.

2.2 Assumptions and Dependencies

Assumptions:

1. Stable Internet connection: The system assumes users will have a stable internet connection for smooth and real-time interaction with the API and backend.
2. Token system Integrity: It is assumed that the token management system will accurately deduct and add tokens without any errors.
3. User device compatibility: the system assumes that the users will access the editor in modern web browsers.
4. Blacklist word submissions: It is assumed that the words or ideas included in the blacklist are exempt from being shown when content is generated from the system.
5. Authentication system: Users in their respective roles are functional and secure. When a payment is made for a user, the gateway is secure, and once a payment is confirmed the system correctly updates the user’s status.

6. Data storage: All data relating to users' submissions, corrections, reports, and accounts are maintained and stored safely to protect and back up when necessary.

Dependencies

1. Open AI API: The system relies on OpenAI to generate text and grammar corrections.
2. Flask backend: The backend logic and API endpoints depend on Flask, it handles authentication, token management, and communication with the AI models.
3. AWS RDS: The system depends on the AWS RDS to store user data, tokens, and blacklisted words.
4. Rate-limiting service: to prevent abuse, the server relies on the rate-limiting mechanism which restricts the amount of requests users can send per minute.

3. Specific Requirements

3.1 Use-Case Reports

Use Case: User Authentication

Actors: Free Users, Paid Users, Super Users

Description: Users authenticate using a login system that differentiates roles.

Preconditions: Users must have valid credentials (except for Free Users who do not require login).

Flow of Events:

1. User enters username and password (if applicable).
2. System verifies credentials.
3. Users are granted access based on their role.

Postconditions: User gains appropriate access rights.

Use Case: Text Submission

Actors: Free Users, Paid Users

Description: Users submit texts for LLM-based correction.

Preconditions: Users must be logged in (except for Free Users), and Paid Users must have enough tokens.

Flow of Events:

1. User inputs text in the editor.
2. System checks word count (limit for Free Users, token calculation for Paid Users).
3. If within the limit, the text is processed; otherwise, the submission fails with penalties.

Postconditions: Text is submitted for correction.

Use Case: Text Correction

Actors: Paid Users, LLM

Description: Users choose self-correction or LLM-assisted correction.

Preconditions: Submitted text must be available.

Flow of Events:

1. User selects the correction method.

2. If self-correction, the system deducts half the required tokens.
3. If LLM correction, AI highlights errors for user approval.
4. User accepts/rejects corrections.

Postconditions: The corrected text is updated.

Use Case: Token Management

Actors: Paid Users

Description: Paid users manage their tokens.

Preconditions: User must be logged in as a Paid User.

Flow of Events:

1. User checks token balance.
2. User purchases additional tokens if needed.
3. System updates token balance.

Postconditions: Token balance reflects transactions.

Use Case: Blacklist Management

Actors: Free Users, Paid Users, Super Users

Description: Users suggest blacklist words; Super Users approve/reject them.

Preconditions: User must be logged in.

Flow of Events:

1. Free/Paid Users submit words for blacklisting.
2. Super Users review and decide.
3. System updates blacklist.

Postconditions: Blacklist is updated accordingly.

Use Case: Complaint Handling

Actors: Paid Users, Super Users

Description: Users submit complaints about collaboration or system issues.

Preconditions: User must have a dispute.

Flow of Events:

1. Paid User files a complaint.
2. Super User reviews the case.
3. Super User issues a resolution (penalty, warning, dismissal).

Postconditions: Complaint is resolved.

See Appendix B.

3.2 Supplementary Requirements

Usability requirement

- User Interface: The editor needs to be organized in a clean and user-friendly manner with simple navigation.
- Error Handling: Display specific error messages upon invalid submissions or lack of tokens.
- Mobile Compatibility: Provide a responsive UI to mobile and tablet users.

Compatibility Requirements

- Browser Support: The editor must support Firefox, Chrome, and Safari.
- API Compatibility: The backend API must be RESTful and OpenAI model compatible.

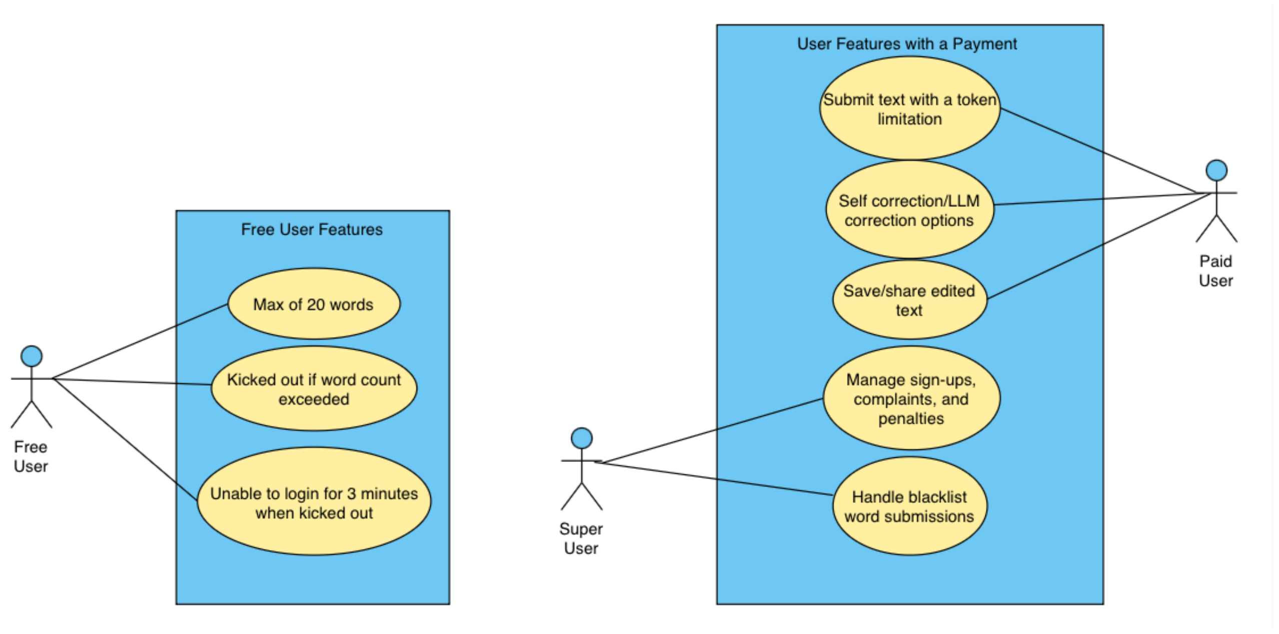
4. Supporting Information

Appendices:

1. Appendix A: File Structure of the Project

```
flask-app/
│
├── LLM/                    # Main application module
│   ├── __init__.py         # Initializes the Flask app and database
│   ├── routes.py           # Defines the routes (login, signup, logout, home)
│
├── templates/              # HTML templates for the frontend
│   ├── base.html           # Base template for all pages
│   ├── index.html          # Home page
│   ├── login.html          # Login page
│   └── signup.html         # Signup page
│
├── static/                 # Static files (CSS, JS, images)
│   └── styles.css          # Styles for the frontend
│
├── app.py                  # Main entry point for the Flask application
└── requirements.txt        # Lists Python dependencies
```

2. Appendix B: Use Case Diagram


Free User Features
Max of 20 words
Kicked out if word count exceeded
Unable to login for 3 minutes when kicked out
Free User
User Features with a Payment
Submit text with a token limitation
Self correction/LLM correction options
Save/share edited text
Manage sign-ups, complaints, and penalties
Handle blacklist word submissions
Paid User
Super User

3. Appendix C: Creating a model for the user schema

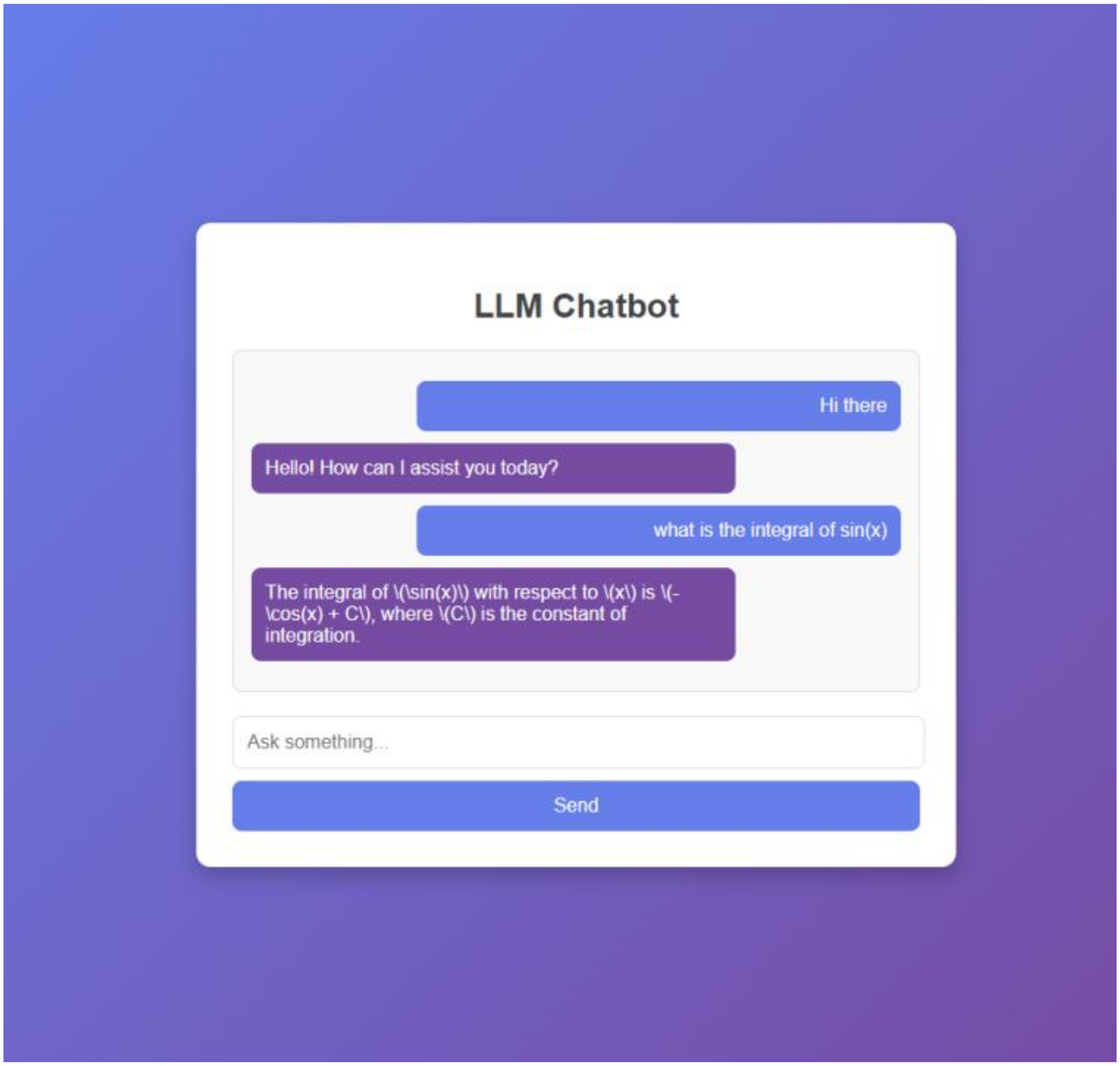
```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password_hash = db.Column(db.String(128), nullable=False)
    Tokens = db.Column(db.Integer, unique=True, nullable=False)
```

Login Route

```
@bp.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')
        user = User.query.filter_by(email=email).first()
    if user and user.check_password(password):
        session['user_id'] = user.id
        flash('Login successful!', 'success')
        return redirect(url_for('main.home'))
    else:
        flash('Invalid email or password', 'error')
return render_template('login.html')
```

Endpoint	Method	Parameters	Response	Description
/	GET	None	Renders index.html	Displays the home page
/login	GET, POST	Email, password	Renders login.html or redirects to /	Handles user login.
/signup	GET, POST	Email,username,password	Renders signup.html or redirects to /	Handles user registration.
/ask	POST	Query	Renders the answers for a specific question using OpenAI API	Sends back answers to query

4. Appendix D: Prototype (UI)



Hi there

Hello! How can I assist you today?

what is the integral of sin(x)

The integral of \(\sin(x)\) with respect to \(x\) is \(-\cos(x) + C\), where \(C\) is the constant of integration.

Model for the text editor page that paid and free users are greeted with after completing sign up.

Index:

Term	Section	Description

Flask	1.3, 1.4	A Python web framework used to build the application.
Flask-SQLAlchemy	1.3, 1.4	An ORM for interacting with the database in Flask applications.
AWS RDS	1.3, 1.4	Amazon Relational Database Service used for hosting the MySQL database.
MySQL	1.3, 1.4	The relational database management system used in the project.
Jinja2	1.3, 1.4	The templating engine used by Flask to render HTML templates.
Werkzeug	11.3, 1.4	A WSGI utility library used by Flask for handling HTTP requests and security.
User Authentication	1.3, 2.0	The process of verifying user identity through login and signup functionality.
Session Management	1.3, 2.0	The mechanism for maintaining user state across multiple HTTP requests.
Static Files	1.3, 2.0	Files like CSS and JavaScript served directly to the client.
Routes	1.3, 2.0	URL handlers that define the behavior of the application.