SUGARLABS

ADD AN AI ASSISTANT TO THE WRITE ACTIVITY

March 26, 2025

T ASWATH
aswathscid@gmail.com
github.com/t-aswath

Contents

About Me

Hi, I'm T ASWATH, a 3rd-year B.E. CSE student at Chennai Institute of Technology with a deep passion for AI, open-source, and competitive programming. My programming journey began in 12th grade, and since then, I have explored diverse technologies, from system design to cloud computing.

- Email: aswathscid@gmail.com
- Github: t-aswath
- Location: India (IST)
- First Language: Tamil

Previous Works

Open Source

Organization	Pull request	Issue
Sugar Labs	2 merged, 1 open	4 closed, 1 open
LibreOffice	3 merged	-
OhMyZsh	1 merged	-

Table 1: Contributions to Open Source Projects

Meetings

- Activity Team/Meetings/2025-03-12 - Intro/ phrase 1 of prototype
- Activity Team/Meetings/2025-03-19 - phrase 2 of prototype
- Activity Team/Meetings/2025-03-23 - phrase 3 of prototype
- Activity Team/Meetings/2025-03-26 - progress on phrase 4 of prototype

Bitspace

I cofounded a **student-led open source organization** during my first year of college, together with my friends. Currently, I serve as the **Vice President**. Our organization conducts **workshops, hosts hackathons, and organizes competitions** focused on open-source development.
Our mission is to educate students about the significance of open source and to encourage their active participation in the community.

Recent Activities

- **Community Partner** Local Host: Chennai (CIT) at **FossHack 2025** by **FossUnited**
- Hosted a **booth** at the **INDIA FOSS Conference 2024** by **FossUnited** (Check out the booth list)

Projects

I have worked on **4 RAG-based projects**—one during my internship at **Cognizant** and three for **Microsoft Innovation Challenge**, where one of my projects secured **3rd place (TUSK)** in the competition.

- https://github.com/t-aswath/TUSK
- https://github.com/bitspaceorg/trusted-utility-for-statutory-knowledge-act-ii
- https://github.com/bitspaceorg/smart-process-innovation-network

I am proficient in technologies such as:

- **Python**
- **Gtk**
- **LangChain**
- **ChromaDB**
- **AWS**
- **Ollama**
- **Hugging Face**
- **Docker**
- **OpenAI**
- **RAG**
- **JS**
- **TS**
- **REST API**
- **GIT**
- **Prompt Engineering**

These skills are particularly valuable for this project. For more details on my other skills, check out my **Resume** and **GitHub**.

Availability

By the end of April, my end semester exams will be over, and I will move on to my final year. At my college, there are **no coursework requirements** in the final year, as all subjects are completed in the previous semesters. The final year is entirely dedicated to internships.
My college is familiar with the **Google Summer of Code** program, thanks to past contributors, and provides **full-time support for students** to work on GSoC.

Project Details

Name: Add an AI assistant to the Write Activity

Description: Sugar pioneered peer editing in its Write Activity. However, the Write Activity has never had any serious support for grammar correction (just spell check) and none of the more recent developments around AI-assisted writing. The goal of this project is to add AI assistance to the writing process: both in the form of providing feedback as to what has been written and making suggestions as to what might be written.
Project Length: 350 hours
Difficulty: High
Coding Mentors: Walter Bender, Ibiam Chihurumnaya

Solution

Project Overview

The goal of this project is to develop a real-time AI-powered writing assistant for the Write activity in Sugar. This assistant will help children improve their grammar by not only identifying and correcting mistakes but also explaining why a correction is needed and how it improves their writing.

To achieve this, the system will extract text from the Write activity, process it using a language model (LLM), and return the corrected text along with detailed explanations. These insights will be presented in a simple, intuitive, and engaging user interface, ensuring a seamless learning experience. The focus is on enhancing children's writing skills interactively and educationally, making grammar correction a learning opportunity rather than just an automated fix.

Deliverables

- **Two Grammar Checking Modes:**
  - **Co-Pilot Mode**: Provides real-time grammar suggestions as the child writes, acting as a supportive writing assistant.
  - **Test Mode**: Allows children to write freely and receive corrections only after completing their text, encouraging independent learning.
- **Intuitive User Interface:**
  - A user-friendly UI that highlights corrections, making it easy for children to understand and learn from their mistakes.
- **Grammar Checker Status Indicator:**
  - A visual indicator that displays the **real-time status** of the grammar checker, ensuring users are aware of when corrections are being processed.
- **Auto-Complete for Corrections:**
  - An intelligent **auto-complete** feature that seamlessly integrates suggested corrections into the text, helping children learn proper grammar effortlessly.
- **Suggestions:**
  - Grammar suggestions will be tailored to specific **age groups**, ensuring **clarity and ease of understanding**. The complexity of suggestions will be adjusted based on the user's age to provide appropriate and effective guidance.

DESIGN

I am proposing only the layout of the interface, not the specific style or color scheme, as these aspects are best discussed with the UI/UX team to ensure alignment with Sugar Labs' design standards. I have outlined two possible layout options, but we can proceed with either of these or adopt an entirely different layout if Sugar Labs already has a preferred design in mind. I am fully flexible and open to implementing any design that best fits the project's needs.

Design - I

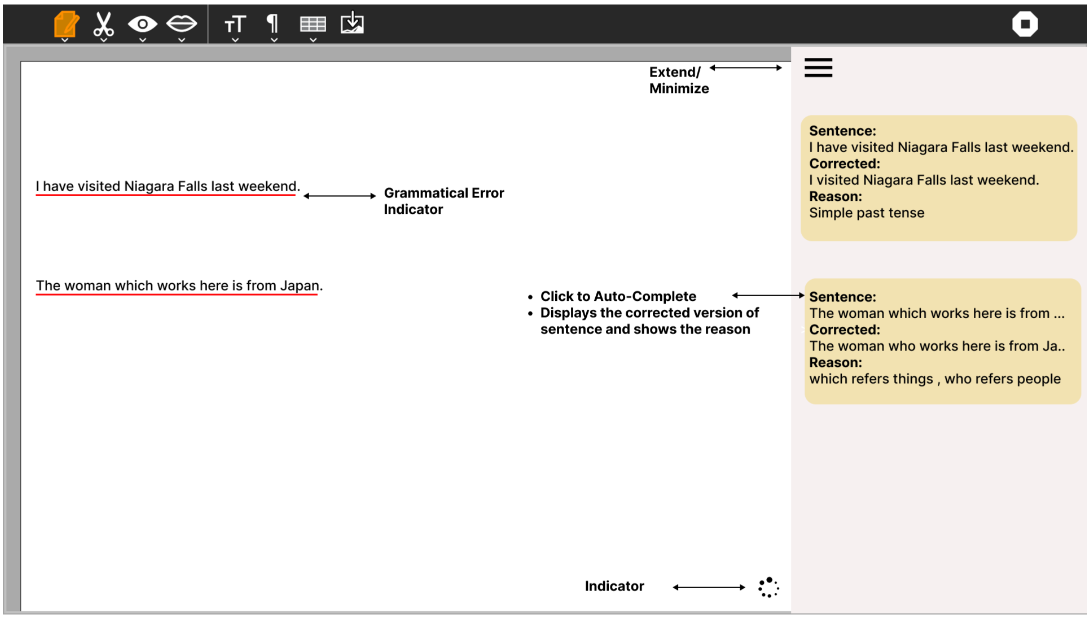

Figure 1: Sidebar style layout

Design - II

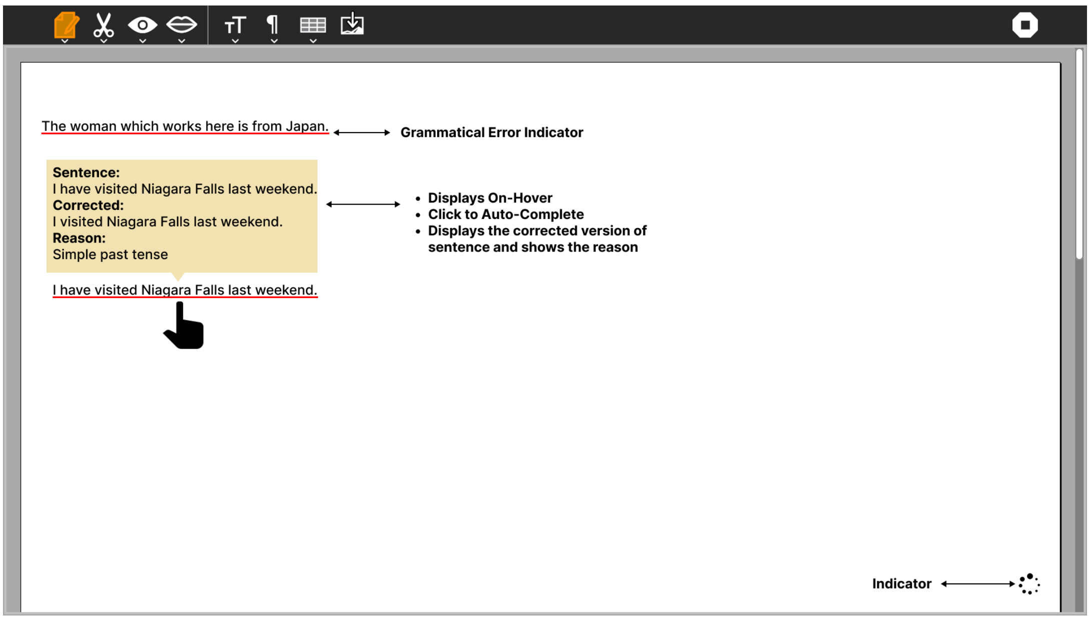

Figure 2: Tool Tip style layout

Indicator

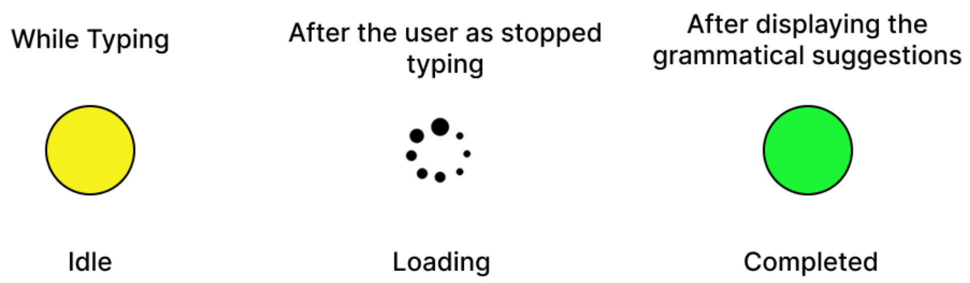

Figure 3: Indicator and its States

Test Mode

Figure 4: Button to switch modes

TECH STACK

- **Python**
- **Asyncio**
- **Ollama**
- **LangChain**
- **pydantic**
- **Docker**
- **FastAPI**
- **Gtk**
- **AWS**

IMPLEMENTATION

This section provides a comprehensive breakdown of the project's implementation, divided into several key components.

User Interface (UI)

For the UI, we will adhere to the traditional Sugar Labs approach by leveraging **GTK** with **sugar3**, specifically:

- `sugar3.graphics`
- `sugar3.activity.widgets`

Additionally, we will incorporate custom widgets from `widgets.py` along with newly designed widgets to implement the proposed UI design.

Required UI Components

The following widgets will be essential for the project:

- **Sidebar** – Displays grammar suggestions.
- **Suggestion Widget** – Shows suggested corrections.
- **Sidebar Toggle Button** – Allows users to open and close the sidebar.
- **Mode Toggle Button** – Switches between different modes (Test, Copilot).
- **Progress Indicator (Custom Spinner)** – Displays the status of the grammar checker.
- **Custom Text Tags** – Highlights errors and displays suggestions.
- **Tooltip-like Suggestions** – Provides inline grammar suggestions.

Text Highlighting

To highlight errors within the text:

1. We will create custom `Gtk.TextTag` objects for text highlighting.
2. Upon receiving the payload, the erroneous sentences will be identified.
3. The identified errors will be highlighted using these custom tags.

Sidebar Design

The sidebar will contain:

- A `Gtk.ScrolledWindow` to display multiple suggestions.
- `Gtk.Label` elements to present grammar suggestions.
- `Gtk.EventBox` to make suggestions clickable, enabling automatic corrections.

Styling

To ensure the UI aligns with Sugar Labs' design principles, we will use **CSS** for styling.

- `Gtk.StyleContext` and `Gtk.CssProvider` will be utilized to apply custom styles to the widgets.
- The UI will be **clean, responsive, and visually appealing**, ensuring an intuitive experience for users.

Frontend

The frontend, or the **Activity's back office**, is responsible for managing the core functionalities of the project. It handles user interactions, processes text modifications, communicates with the backend, and ensures efficient performance without compromising user experience.

Core Responsibilities

The frontend is responsible for handling the following features:

1. **Event Listeners for the Text View**

- Detects the right time to request grammar suggestions.
- Monitors text modification events.
- Implements **debouncing** and **throttling** to optimize request timing.
- Calculates **user pauses** to determine when to request suggestions.

2. **Requesting Suggestions from the Backend**

   - Sends the modified text along with the user's **age group** to receive **age-appropriate** suggestions.

3. **Handling Backend Responses**

   - **Highlights errors** in the text.
   - **Marks text for auto-correction**.
   - **Stores suggestions** for easy access.
   - **Applies auto-corrections** when suggestions are clicked.
   - **Displays suggestions** in the sidebar.

Event Listeners in the Text View

To efficiently determine when to request suggestions, event listeners will track various actions that modify the text, including:

- **Text modification events** (`insert-text`).
- **Keystrokes** (`backspace`, `delete`, `enter`).
- **Editing actions** (`paste`, `cut`, `undo`, `redo`).

These listeners help detect:

- **User pauses** (to trigger requests when typing slows down).
- **Sentence/paragraph completion** (to request suggestions at logical points in the text).

By monitoring these interactions, we ensure that suggestions are requested only **when necessary**, improving efficiency and reducing unnecessary backend requests.

Optimizing Requests with Debounce and Throttle

To prevent excessive backend calls and ensure performance efficiency, we will implement:

- **Debouncing** – Cancels a previous request if a new request is made before the previous one is completed. This prevents redundant requests when users are actively typing.
- **Throttling** – Limits the number of requests made within a specific time frame, ensuring that the backend is not overloaded with frequent requests.

The frontend will send requests using the `requests` library, providing:

- The **text to be checked**.
- The **user's age group** (to tailor grammar suggestions accordingly).

Processing Backend Responses

Once the backend returns grammar suggestions, the frontend will:

1. **Locate the Sentences Containing Errors**
   - The response will include a list of **problematic sentences** and their respective **corrections**.
   - The frontend will search for these sentences in the **Gtk.TextBuffer** using the `forward_search` method.
2. **Highlight the Incorrect Text**
   - Custom `Gtk.TextTag` objects will be created for **highlighting** text with errors.
3. **Mark Text for Auto-Correction**
   - **Gtk.TextMark** will be used to track portions of text flagged for **correction**.
   - These marks allow quick retrieval of text positions using **get_iter_at_mark**.
4. **Store Suggestions for Quick Access**
   - All received **corrections** will be stored in a **dictionary**, ensuring efficient retrieval.
5. **Display Suggestions in the Sidebar**
   - Suggestions will be dynamically displayed using **custom widgets** inside the sidebar.
   - Clicking a suggestion will trigger **auto-correction** (explained below).

Auto-Correction Mechanism

When a user **clicks on a suggestion**, the incorrect text will be replaced using `Gtk.TextBuffer` methods:

- **delete()** – Removes the incorrect text.
- **insert()** – Inserts the corrected version.

Once the correction is applied:

- The suggestion will be removed from the sidebar.
- The corresponding text highlight will be **cleared**.

Ensuring UI Responsiveness

To prevent the UI from freezing when processing large amounts of text, we will:

- Use **GLib.idle_add()** to execute functions within the main loop asynchronously.
- This ensures that the text processing and UI updates happen **smoothly** without blocking user interactions.

Efficient Sentence Searching

Searching for problematic sentences in long text documents can be computationally expensive. **Native Gtk search methods operate in O(n) time**, meaning that searching for n sentences takes **$O(n^2)$** time.

To optimize this:

1. **Limit the Search Range**
   - Since we process text incrementally, we can **restrict the search scope** to newly modified text instead of scanning the entire document.
2. **Leverage Write Activity's Search Functionality**
   - If **Write Activity's** built-in search mechanism provides a more efficient algorithm, we will integrate it.
   - If its performance is **comparable to Gtk's**, we will adapt our optimization strategies to improve it.

Integration with AbiWord for Contextual Grammar Checking

To improve efficiency further, we will:

1. **Detect the User's Current Page**
   - AbiWord provides tools to determine which **page** the user is currently working on.
2. **Limit Suggestions to Relevant Pages**
   - Instead of checking the **entire document**, we will restrict grammar checking to:
     - The **current page**.
     - **One or two pages before and after** the current page.
3. **Cache Suggestions for Each Page**
   - By caching **suggestions per page**, we:
     - Avoid **repeated backend requests**.
     - Allow users to **switch pages seamlessly** without reloading suggestions.

By implementing these strategies, we ensure a **fast, accurate, and user-friendly grammar correction system** while optimizing **computational resources**.

Backend

Now, let's dive into the **backend**, which I believe is the **heart of this project**. The backend is responsible for processing user input, performing grammar checks, and returning structured suggestions to the frontend.

Core Responsibilities

The backend will handle the following key functionalities:
POST "/invoke" Endpoint

- Receives the **text** to be checked along with the **user's age group**.

Grammar Checking using an LLM

- Uses a **Large Language Model (LLM)** to identify grammar mistakes.

Text Rephrasing and Formatting

- Improves text clarity and readability, making it easier to understand.

Response Validation and Type Checking

- Ensures all responses follow a well-defined structure using `pydantic`.

Logging and Debugging

- Maintains detailed logs for monitoring and debugging purposes.

Sending the Response to the Frontend

- Delivers structured and validated results to the frontend.

Implementation Details

The backend will be built using **FastAPI**, a modern web framework that provides high-performance APIs with built-in support for type validation and asynchronous processing.

We will create a **POST** endpoint `/invoke`, which will:

- Accept **text input** and **user age group**.
- Validate incoming data using `pydantic`.
- Invoke a chain of functions that:
  - **Analyze grammar mistakes** using an LLM.
  - **Rephrase and format the text**.
  - **Validate and structure the response**.
- Return the processed data to the frontend.

Processing Flow: The LangChain Workflow

The AI-driven workflow follows this structured **LangChain** pipeline:

Step 1: Receive User Input

- The input text is formatted for processing.
- A structured **prompt** is prepared for the **LLM** to analyze grammar mistakes.

Step 2: Grammar Checking using an LLM

- The **LLM** is called to detect grammatical issues.
- The model follows a predefined **system prompt** to analyze the text.
- The response includes:
  - **Grammar errors detected**.
  - **Suggestions for correction**.

Step 3: Rephrasing and Formatting

- The response is further processed to improve clarity and readability.
- This ensures that users of **any literacy level** can easily understand the corrections.

Step 4: Send the Response to the Frontend

- The final response is structured and sent back to the frontend.

Each step includes **response validation** to ensure correctness.

Ensuring Scalability and Performance

Asynchronous Processing

- To handle multiple requests efficiently, all functions will use **async** and **await**.
- This prevents blocking operations and improves response time.

Leveraging LangChain for AI Processing

- We will utilize **LangChain's asynchronous methods** like:
  - `ainvoke()` – Asynchronously invokes the LLM.
  - `abatch()` – Processes multiple requests in parallel.
- This ensures **optimized performance** when handling multiple users simultaneously.

Handling JSON Responses and Fixing Errors

- Since the LLM generates responses in **JSON format**, we will use the `json` module to process them.
- However, models sometimes produce **malformed JSON** due to syntax errors.

- To fix this, we will implement `OutputFixingParser` from LangChain:
  - It **automatically corrects syntax errors** in the JSON response.
  - Ensures that the final output is **structured correctly** before sending it to the frontend.

Model Execution Using Ollama

To run the **LLM**, we will use **Ollama**, a framework that allows us to deploy and manage AI models locally or on a server.

- We will **pull the required model** into our environment.
- Use the `langchain-ollama` integration to execute the model within our FastAPI backend.
- This provides efficient, **asynchronous AI processing** with minimal latency.

Model Selection and Processing

In this section, we will discuss the **process of model selection** and the **methods** used to achieve the expected results. The following key topics will be covered:

1. Model Selection
2. Grammar Checking
3. Rephrasing
4. OutputFixingParser for JSON Correction

1. Model Selection

Selecting the right model is a **critical step** in ensuring the effectiveness of the grammar-checking system. We will explore various sources such as:

- **Hugging Face** – A repository of state-of-the-art NLP models.
- **Ollama Collections** – Provides LLMs optimized for local inference.
- **Kaggle** – Offers access to fine-tuned and pre-trained models.

We will evaluate two types of models:

1. **General-purpose language models** – These can be **fine-tuned** for grammar correction.
2. **Pre-trained grammar-checking models** – Models specifically trained for **grammatical analysis and correction**.

Benchmarking and Evaluation Criteria

To identify the best model for our project, we will conduct rigorous benchmarking based on the following factors:

- **Accuracy** – How well the model identifies grammar mistakes.
- **Inference Speed** – The response time for real-time grammar checking.
- **Context Window Size** – The model's ability to analyze large text inputs.
- **Reasoning Ability** – The model's capacity to understand complex sentence structures.
- **Response Quality** – The clarity and correctness of the model's output.

The selected model will be integrated into our **FastAPI backend**, ensuring **scalability and efficiency**.

2. Grammar Checking

For accurate grammar correction, a well-crafted **system prompt** will be used to instruct the model.

Grammar Checking Process

1. The user submits text input.
2. The backend prepares a structured **system prompt** that:
   - Defines grammar rules.
   - Requests suggestions in a **JSON format**.
   - Ensures the output includes both **error explanations** and **corrections**.
3. The **LLM processes** the input and returns a structured response.
4. The backend validates the response before sending it to the frontend.

By optimizing **prompt engineering**, we ensure that the model provides **highly accurate and context-aware grammar suggestions**.

3. Rephrasing for Better Understanding

To make the suggestions more user-friendly, **rephrasing and formatting** will be applied. This ensures that **users of all literacy levels** can easily understand the feedback.
Rephrasing Strategy

- We will use **template-based prompting** to instruct the model on how to rewrite suggestions.
- The **ChatPromptTemplate** from LangChain will be used to structure the input prompts.
- The rephrased output will follow a **concise yet clear** format.

By incorporating **context-aware rephrasing**, the feedback becomes more **accessible and actionable** for users.

4. OutputFixingParser: Ensuring Valid JSON Responses

Since the model **generates JSON responses**, occasional **syntax errors** can occur. To handle this, we will integrate `OutputFixingParser` from LangChain.
How OutputFixingParser Works

1. The raw JSON response from the model is **parsed**.
2. If a syntax error is detected, `OutputFixingParser`:
    - **Identifies the error**.
    - **Fixes missing/incorrect syntax**.
    - **Returns a corrected JSON output**.
3. The backend validates the corrected JSON before sending it to the frontend.

This approach ensures that **even if the LLM generates a malformed response**, it will be **automatically corrected** without affecting system performance.

Integration into Backend

All these components—**grammar checking, rephrasing, and JSON validation**—will be implemented within the **backend** as part of a **chain of functions**.

By leveraging **FastAPI, LangChain, and Ollama**, we create an **efficient, accurate, and scalable** grammar-checking system that provides high-quality suggestions with **minimal processing time**.

Logger (Optional but Recommended)

Logging can be a valuable addition to the project, ensuring **performance monitoring, debugging, and adherence to Responsible AI guidelines**. While we **will not log sensitive user data**, such as text inputs or model responses, we can track essential metadata to improve system efficiency and reliability.

Key Data Points to Log

To monitor the system's performance and detect potential issues, we can log:

- **API call duration** – Time taken for each request to be processed.
- **Number of requests** – Helps track usage patterns.
- **Error rates** – Identifies recurring failures or bottlenecks.
- **Success rate** – Measures the accuracy and efficiency of the model.

Recommended Logging Methods

LangSmith LangSmith is a specialized logging library designed for **AI workflows and LangChain models**. It integrates seamlessly with LangChain pipelines and provides detailed **insights into model performance, API calls, and errors**. Key advantages include:

- **Easy integration** using function decorators.
- **Structured logging** for tracking AI interactions.
- **Visualization and monitoring** of model behavior.

Python's Built-in `logging` Module The **Python `logging` module** is a lightweight, flexible, and customizable option for logging metadata. It supports various **logging levels (INFO, DEBUG, ERROR, WARNING, CRITICAL)** and allows:

- **Simple implementation** with minimal dependencies.
- **Custom log formatting** to capture key details.
- **Storage flexibility**, including console output or file logging.

By implementing an **efficient logging system**, we can enhance **observability, troubleshoot issues faster, and optimize system performance**, ensuring a robust and scalable solution.

Deployment

For deployment, we will use an **ASGI server** recommended by FastAPI DOCS, such as **Uvicorn** or **Hypercorn**, to run the backend efficiently.

The backend will follow the structure of **Sugar-AI** and be **containerized using Docker** for portability and scalability. We will utilize the **Ollama container from Docker Hub** to streamline AI model management.

The system can be deployed on **cloud platforms (AWS, Azure, GCP), virtual machines, or local environments**, ensuring flexibility and ease of maintenance.

Tests (Optional)

Given the extensive range of activities in **Sugar-AI**, maintaining **code quality** across all functionalities can be challenging. To ensure the system operates reliably and meets project requirements, implementing **automated testing** can significantly reduce the burden of **manual verification** while improving overall stability.

Since the codebase for individual activities is relatively small, it can be efficiently tested. We will utilize **pytest**, a powerful and flexible testing framework, to write **unit tests** and **integration tests**. Unit tests will verify the functionality of isolated components, while integration tests will ensure smooth interaction between different parts of the system.

By incorporating a **structured testing approach**, we can enhance **code reliability, detect issues early, and streamline the development process** while maintaining high performance across all activities.

Architecture

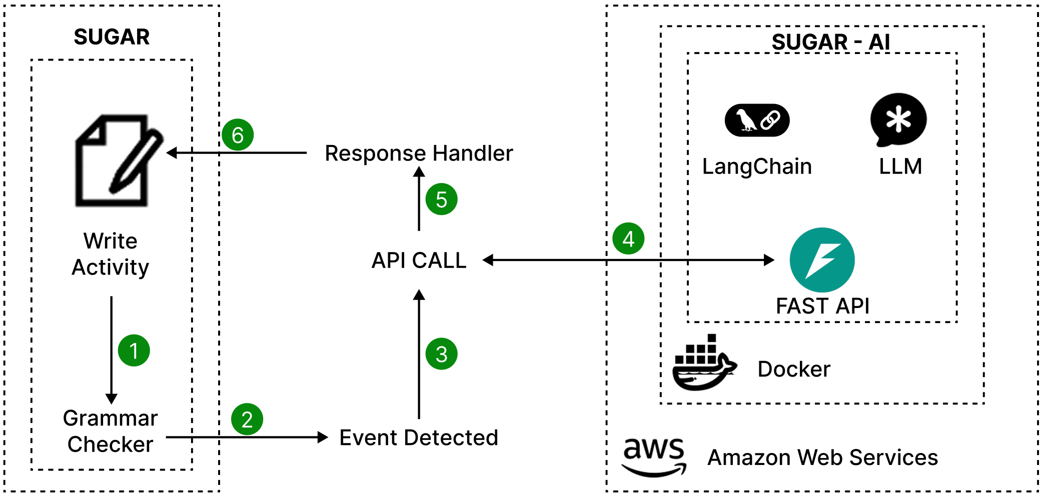

Figure 5: Architecture

Prototype

The prototype's development journey is thoroughly documented in the project's GitHub README, detailing its architecture, functionality, and implementation process. Incremental progress is transparently showcased through commit history and demonstration videos, providing a clear view of its evolution.

Code Repository: GitHub - sugarlabs-proto

Development Iterations:

- **Iteration 1:** Watch on YouTube
- **Iteration 2:** Watch on YouTube
- **Iteration 3:** Watch on YouTube

Impact

1. Enhancing Learning for Children

This project will **transform** the way children interact with the **Write** activity by providing real-time **grammar correction and explanations**. Instead of simply fixing errors, it will:

- **Encourage learning** by explaining why a correction is needed.
- **Improve writing skills** through interactive suggestions.
- **Boost confidence** in language proficiency by offering a supportive AI-powered assistant.

By making grammar correction a **learning experience**, children will develop stronger writing skills in a fun and engaging way.

2. Strengthening Sugar Labs' Educational Tools

This project aligns with Sugar Labs' mission of **enhancing digital learning** by:

- Expanding the capabilities of the **Write** activity.
- Introducing a **modular AI component** that can be reused across other activities.
- Improving accessibility and inclusivity by helping children **of all literacy levels** enhance their writing skills.

3. Advancing Open-Source AI in Education

By integrating **AI-powered language processing** into Sugar Labs, this project will:

- Contribute a **scalable, reusable AI module** to the open-source community.
- Demonstrate how **AI can be used for education** in a way that prioritizes learning over automation.
- Set the foundation for future AI-powered educational tools within Sugar.

By bridging the gap between **AI and education**, this project will have a **lasting impact** on how children learn and interact with digital tools within Sugar Labs.

Post-GSoC Plans & Future of the Project

1. Long-Term Sustainability & Maintenance

After GSoC, I plan to ensure the project remains **maintainable, scalable, and adaptable** by:

- **Actively contributing** to the Sugar Labs community to refine and improve the AI module.
- **Providing thorough documentation** to make it easier for future developers to enhance the system.

2. Future Enhancements & Improvements

While the current implementation focuses on **real-time grammar correction and learning**, there is immense potential to expand the project further:

1. **Multilingual Support**
   - Expanding beyond English to assist children in learning and improving multiple languages.
   - Incorporating translation and grammar correction for various linguistic backgrounds.
2. **Adaptive Learning**
   - Implementing an **AI-driven personalized learning** experience that adjusts suggestions based on a child's writing habits.
   - Introducing **difficulty levels** to help students progressively improve their skills.

3. My Continued Contribution

I am committed to continuing my involvement with Sugar Labs even after GSoC by:

- **Helping onboard new contributors** to maintain and expand the AI module.
- **Optimizing the AI model** by experimenting with new LLMs and fine-tuning for better accuracy.

This project is just the **beginning of AI-powered learning** within Sugar Labs. With continuous enhancements, it has the potential to become a **core educational tool** that empowers children worldwide to improve their writing skills in a fun and interactive way.

TIMELINE

Community Bonding Period

- Get familiar with the **Sugar Labs ecosystem**, its codebase, and development workflows.
- Engage with **mentors and contributors**.
- Finalize the technical design and discuss any modifications with the community.
- Set up the development environment and define test cases for grammar correction.

Week 1-2: Initial AI Module Setup & Experimentation

- Explore and benchmark various **LLMs** for grammar correction.
- Integrate **LangChain** and structure the AI pipeline.
- Implement a **basic API using FASTAPI** for grammar correction.
- Conduct **unit testing** to validate LLM responses.

Week 3-4: API Development & Dockerization

- Finalize the **best-performing LLM** for grammar correction.
- Implement a **robust FASTAPI route** to handle text correction requests.
- **Dockerize** the AI module for easy deployment and integration.
- Set up **logging mechanisms** to track API usage and interactions.
- Conduct **initial API testing** to ensure stability.

Week 5: Midterm Evaluation & UI Integration Begins

- Submit work for **midterm evaluation**.
- Start integrating the **Write** activity with the AI module via API calls.
- Implement **event listeners** in the text area to capture user activity.
- Develop a **debouncing mechanism** to prevent redundant API calls.

Week 6-7: UI Improvements & Real-Time Feedback

- Apply **styles and highlights** to mark grammar mistakes.
- Display **suggestions and reasoning** within the Write activity.
- Implement **interactive elements** for user engagement (e.g., tooltips, pop-ups).
- Conduct **integration testing** between the Write activity and AI module.

Week 8: Advanced Features & Performance Optimization

- Implement an **interception mechanism** to cancel redundant backend computations.
- Optimize **response times** and improve model efficiency.
- Refine the **auto-complete feature** to suggest grammatically correct text.
- Test **UI responsiveness** across different screen sizes.

Week 9: Extensive Testing & Bug Fixes

- Perform **end-to-end testing** covering unit, API, and UI functionality.
- Gather feedback from **mentors and the community** for final improvements.
- Fix bugs, optimize performance, and polish the overall experience.

Week 10: Final Refinements & Documentation

- Finalize all **code, documentation, and tutorials**.
- Submit the project for **final evaluation**.
- Write a **blog post/demo** showcasing the project's impact.
- Discuss **post-GSoC plans** with the community for future improvements.

RESEARCH

Understanding Grammarly's Approach

To gain insights into how Grammarly achieves **effective grammar correction with minimal input lag**, I studied the following resources:

- ACL Anthology: Automated Grammar Error Correction
- Grammarly's Engineering Blog: Reducing Text Input Lag
- How Grammarly Works
- Grammarly's NLP Approach for Run-on Sentences
- How Grammarly Uses AI

Building a Custom Grammar Correction Tool

To develop our own AI-driven grammar correction tool, I researched methodologies and best practices:

- How to Build a Grammar Checker Like Grammarly
- Creating a Custom AI-Based Grammar Checker

Existing Grammar Correction Tools & Comparisons

To evaluate different approaches and understand their strengths, I reviewed various existing tools:

- Ginger Software
- Zoho Writer Grammar Checker
- GrammarCheck
- TutorBin Grammar Checker

Relevant Past GSoC Projects from Sugar Labs

Studying past GSoC projects provided valuable insights into Sugar Labs' development standards and best practices:

- AI Chatbot Integration for Chat Activity
- Pippy Activity Enhancements
- Sugar Labs GSoC Archive

Technologies

Extensive research was conducted to identify the most suitable methods and functions required to implement the desired functionality effectively. The following technologies played a crucial role in shaping the prototype:

- GTK: – Used for building the graphical user interface.
- LangChain: – Utilized for constructing AI-powered language processing workflows.
- FastAPI: – Enabled the creation of a high-performance API for model interaction.
- Pydantic: – Ensured robust data validation and type enforcement in API requests.
- LangSmith: – Assisted in debugging, monitoring, and optimizing language model pipelines.

CONCLUSION

I am confident in my ability to successfully execute this project and deliver a **high-quality, AI-powered grammar correction tool** for Sugar Labs. With my experience in **AI, LangChain, API development, and system architecture**, I have the **technical expertise** required to build an efficient and scalable solution.

Beyond technical skills, I have a strong background in **open-source development and community-driven projects**, ensuring that my work aligns with **Sugar Labs' values and standards**. My commitment to **clean, maintainable, and well-documented code** will make this project not only impactful in the short term but also **sustainable for future contributors**.

I am highly **dedicated, adaptable, and open to feedback**, ensuring **smooth collaboration** with mentors and the community. My structured approach, **clear milestones, and regular**

progress updates will ensure the timely completion of the project while maintaining high quality.

By integrating **real-time AI-assisted grammar correction**, this project will significantly enhance the **learning experience for children**, empowering them to **write confidently and improve their language skills** in an interactive way. I am excited about this opportunity to contribute to Sugar Labs and make a lasting impact on education through AI.