Project Title:
Kigo

Project Description:
Kigo is a web app that generates haikus using lyrics from the user's most-listened songs. This is achieved by utilizing several APIs to pull the user's listening history, identify the user's most-listened songs, and then retrieve lyrics from these songs. Our algorithms count the syllables and parse these lyrics to create personalized, randomly-generated haikus.

Problem Statement/Customer Value:
Kigo is a fun and interactive program that generates personalized haikus from a user's most-listened songs. This project provides value and entertainment for music lovers by offering a playful and unique way to engage with their listening history. With visually engaging presentations of the haikus, our project combines both technology and art to create a unique, personalized experience for any music enjoyer.

Software Functionality Developed:
Primary Functionality:
- User Authentication: Users authenticate via Spotify, allowing the app to access their listening history.
- Data Retrieval: The app retrieves the user's top tracks using the Spotify API.
- Lyric Fetching: Song lyrics are retrieved for processing using the lyrics.ovh API
- Haiku Generation: Lyrics are processed, parsed for syllable counts, and converted into haikus using algorithms.
- Dynamic User Interface: A React-based frontend allows users to interact with the app, view haikus, and manage preferences.
- Backend Integration: A Node.js backend facilitates communication between the frontend, external APIs, and the Java libraries.

Primary Languages:
- Java: Used for backend logic in the JAR libraries.
- JavaScript: Used for the Node.js backend and React frontend.
- HTML and CSS: Used for frontend design and user interface styling.

Compilers:
- Java Compiler (Maven): For compiling the Java libraries into JAR files.
- Node.js Runtime Environment: Executes the backend and serves the frontend application.

Configuration Management:
1. Spotify Integration: The Spotify Web API was implemented first to handle client IDs, secrets, and authorization tokens.
2. Lyric Fetching: The lyrics.ovh API was integrated to retrieve lyrics for the user's top tracks.
3. Syllable Counting Algorithm: Parallelly, the algorithm to process lyrics and count syllables for haiku generation was developed.

4. Frontend and Backend Integration: The React frontend and Node.js backend were built and connected to the JAR files for processing logic.
5. Testing and Deployment: Final integration and testing ensured adequate operation before deploying the app.

Project Schedules:

Planned Schedule:

(Weeks 1-2) Research & Planning
- Research APIs (Spotify, Genius)
- Define project scope and features

(Weeks 3-6) Development
- Integrate listening history retrieval
- Integrate song lyrics retrieval
- Develop an algorithm to generate the haikus

(Weeks 7-8) User Interface Design
- Design GUI layout and features
- Implement haiku display

(Weeks 9-10) Testing
- Conduct testing
- Gather user feedback for improvements

(Weeks 11-12) Documentation & Finalization
- Prepare user documentation and specifications
- Finalize project for delivery

Actual Schedule:

(Sep 1-3) Project Ideation
- Brainstorm ideas for the project

(Sep 4-5) Research APIs: Kelly, Noah
- Research APIs (Spotify, Genius)
- Define project scope, features, and key functionalities

(Sep 16–20) Set Up Repositories: Sam
- Create GitHub Repository and necessary branches
- Configure repository to use GitHub Pages for static webpage hosting

(Sep 20-Oct 10) Pulling user songs using Spotify API: Noah
- Setup environment with dependencies for Spotify API.
- Develop code to pull top songs from a user.

(Sep 21–Oct 5) Spotify API Integration: Sam
- Integrate Spotify authentication functionality written by Noah into existing GUI
- Test the authentication flow and refine the process

(Oct 6–20) UI Development: Sam
- Design user interface, including decorative elements such as the background, logo, and cursor
- Develop and implement core GUI functionality for navigation and interactivity

(Oct 20-Nov 10) Pull Song lyrics using lyrics.ovh: Noah
- Integrate lyrics.ovh API into environment.

- Develop code to retrieve lyrics for a given song.

(Oct 21–Nov 20) Syllable Counter Development: Kelly

- Write the initial algorithm for syllable counting
- Test and optimize algorithm for accuracy

(Nov 11–20) Haiku Generator: Kelly

- Develop logic to match processed lyric lines to a 5-7-5 syllable pattern and generate haikus
- Test haiku generation for consistency and variability

(Nov 21–29) Integration Front-End/Back-End: Sam

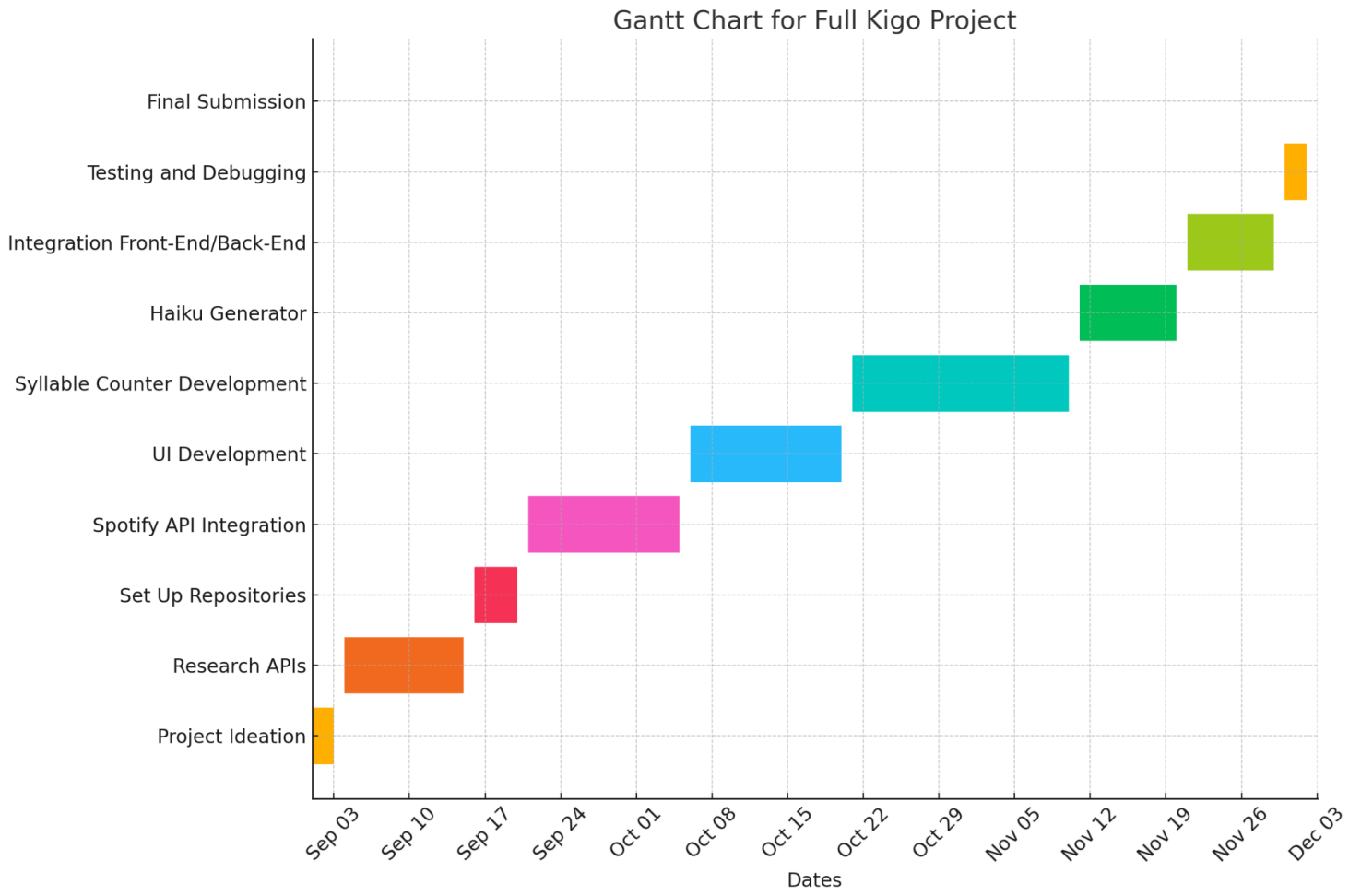
- Connect the Node.js backend to the React frontend
- Set up routes for various tasks such as Spotify authentication, lyric fetching, and haiku generation

(Nov 30–Dec 2) Testing and Debugging: Team

- Conduct unit, integration, and system testing across all components
- Address any remaining issues

(Dec 3) Final Submission: Team

Actual Schedule Gantt Chart:



Engineering Approach:

We used the Scrum framework to guide our software development. We decided this framework would be the best fit for our project because our team is small, and we are working on a fast-paced development project with changing requirements and unknown solutions. Team member's busy schedules and slight difficulty coordinating work simultaneously led to more individual sprints with Sam acting as the Scrum Master directing said sprints.

Software Metrics:

1. **Size**

The Kigo app consists of three main components: the React frontend, the Node.js backend, and the Java libraries which are compiled into JAR files. This LOC (lines of code) analysis was conducted using the statistics utility cloc, excluding blank lines, comments, and auto-generated files.

All source files add up to 78.0 MB.
This does NOT include other software to run code:

- Node.js
- React
- Java
- Maven

The total LOC for the project is 1,526, distributed as follows:

Component	Files Included	LOC
Frontend (React)	All essential files relating to logic and frontend design (app.js, index.js, account.js, app.css)	549 LOC
Backend (Node.js)	Index.js	168 LOC
JAR Source Code	All .java files in kigokelly and kigonoah	kigokelly: 538 LOC kigonoah: 271 LOC Total: 809 LOC

This distribution demonstrates a modular architecture where the frontend accounts for 549 LOC, focusing on user interaction and interface design. The backend, including both the Node.js app (168 LOC) and the Java libraries compiled into JAR files (809 LOC, split between kigokelly with 538 LOC and kigonoah with 271 LOC), totals 977 LOC. This highlights the backend's significant role in handling API integrations, data processing, and haiku generation. The larger LOC allocation for the backend reflects its responsibility for the app's core functionality, while the frontend remains streamlined to ensure a seamless user experience.

2. **Complexity**

Frontend Complexity (React):
The complexity of the frontend is relatively low due to React's component-based architecture. Each file, such as App.js, Account.js, and index.js, is modular, focusing on specific UI functionalities. Conditional rendering is used to adapt the UI based on user actions (ex: logging in or generating haikus). While styling and state management add some complexity, they are handled efficiently through built-in React mechanisms like useState and useEffect.

Backend Complexity (Node.js and Java Libraries):
The backend has moderate complexity due to its dual nature.

- Node.js (Integration Layer):
    - The backend acts as a mediator, connecting the frontend with external APIs and the Java libraries.
    - Routes such as /login and /generate-haiku involve executing external JAR files, which introduces complexity due to the inter-process communication.
    - Error handling for API requests, Spotify authentication, and user data processing is implemented, which adds logical complexity to the index.js file.
- Java Libraries (JAR Files):
    - The libraries perform the heavy lifting of processing Spotify data and generating haikus.
    - The kigonoah library handles API authentication, fetching user top tracks, and retrieving lyrics. Its complexity stems from managing asynchronous calls to the Spotify API and handling edge cases like missing lyrics.
    - The kigokelly library processes the lyrics, calculates syllables, and generates haikus using algorithms. Complexity arises from the rules for syllable counting, handling edge cases (ex: silent "e" in English), and ensuring the randomness of haiku generation.

The integration of the React frontend, Node.js backend, and Java libraries adds another layer of complexity:
- Coordinating data flow between the frontend and backend through API endpoints.
- Managing dependencies and ensuring synchronization between the backend (Node.js) and the compiled JAR files.

Overall, the system achieves a balance of simplicity and functionality. The complexity is concentrated in the backend, particularly in the Java libraries, while the frontend maintains a streamlined and user-friendly design.

3. **Limitations**

Dependency on External APIs
- The Kigo app relies heavily on the Spotify API for user data and the lyrics.ovh API for lyric retrieval. If these APIs experience downtime, rate limiting, or unexpected changes, core functionalities like haiku generation may fail. Additionally, Lyrics.ovh contains a limited database of lyrics and often does not provide lyrics for lesser-known tracks.

Streaming platforms

- This software is only compatible with the music streaming service Spotify. Users who use other streaming services like Pandora, Apple Music, or YouTube Music will not be able to use the software.

Error Handling

- Basic error handling and debugging logs are implemented, which facilitate development. However, user-facing error messages are limited and could be improved to better communicate issues, such as missing data, API failures, or unexpected inputs.

Missing feature implementation

- Some planned features (though non-essential), such as the haiku gallery and additional user preferences are not yet implemented.

4. Quality of Software

The Kigo app demonstrates strong modularity, with a clear separation between the frontend, backend, and Java libraries, making it maintainable and extensible. The source code adheres to standard conventions, ensuring readability and ease of contribution. Basic error handling and debugging logs facilitate development, though user-facing error messages could be improved. The app is efficient for its current scope, leveraging tailored algorithms for haiku generation and responsive React components for a smooth user experience. However, limitations such as dependency on external APIs and missing feature implementation highlight areas for future improvement.

By the standards of ISO/IEC 25010:

Functionality:
All essential functional and non-functional requirements have been met.

Compatibility

- Co-existence - Degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
    - This software can be run locally without having a large impact on the performance of the machine.
- Interoperability - Degree to which a system, product or component can exchange information with other products and mutually use the information that has been exchanged.
    - This software has several parts (frontend development, backend development, Java source code files) which communicate with each other to create a functional web app. This app connects Java source code to perform logic, Multiple APIs to pull data, Node.js, React, and Javascript.

Interaction Capability
Interaction Capability is composed of the following sub-characteristics:

Appropriateness recognizability

- This software is simple and performs one main task- to generate haikus using lyrics from the user's top tracks. Because of this, it is simple for users to determine if this software is appropriate for them.

Learnability

- Disregarding setup, this software is easy to use. There is little complexity to operating this software, as the user only needs to access the web app and connect their Spotify account.

Operability

- The website is intuitive and well designed. This ensures simple navigation and operation of the web app.

User error protection

- Along each step of the process there are checks for errors. For example, in the API sections of the code there is error handling for if a user cannot be authenticated and if lyrics for a song cannot be found.

User engagement-

- Our front end developer (Sam) designed a 2000's inspired retro-style website. This is a unique look for the web app which makes it more memorable for users. Buttons are also thoughtfully placed and ensure easy navigation. Furthermore, haikus are presented in a designated display window for users to view.

Inclusivity

- Since this software is in the early stages of development, it is not yet easily accessible to people who have physical impairments such as blindness. This software is inclusive of anyone who has access to a computer and is comfortable using it.

User assistance

- This software works for everyone given they have a Spotify account and are willing to provide Kigo access to certain user data.

Self-descriptiveness

- This software is very clear in its purpose. An informational summary is provided to users on the homepage of the app, which describes what Kigo does and how it works.

Completion:

Most of the planned primary requirements of the Kigo App are implemented in the product delivered. These include Spotify authentication, fetching the user's top tracks, retrieving lyrics from these top tracks (if possible), processing these lyrics, and generating several haikus. Currently, the Kigo App is functional in terms of these requirements. We planned to implement other capabilities that would make the software more robust and portable, but were unable to fully accomplish these due to time constraints.

The Kigo App is licensed under the MIT license. This means that anyone is free to view, modify, and distribute the source code with minimal restrictions, as long as the original copyright and permission notice are included in all copies or substantial portions of the software. Licensing documentation can be found in the kigo-documentation folder, which is located on the main branch of the Kigo app GitHub repository.

Team members and responsibilities:

Sam Allen: Scrum Master, Front-end Developer, Technical Editor
Noah Kabel: Ethics monitor, Back-end Developer, QA engineer, Technical Editor
Kelly Lowrance: Back-end Developer, Technical Editor, QA Engineer