AI Code generation for lesson plans and model abstraction layer

Ajeet Pratap Singh

Section 1: About us

My name is Ajeet Pratap Singh. I am a final-year undergraduate student at the Chhatrapati Shahu Ji Maharaj University, Kanpur, pursuing a Bachelor’s in Computer Science as my major.

What project are you applying for?

AI Code generation for lesson plans and model abstraction layer

Why are you interested in working with Sugar Labs? And how this project will impact Sugar Labs?

I started contributing to Sugar Labs in November 2023. I started contributing to Sugar Labs because I wanted to explore the open-source community and contribute to projects. Contributing to Sugar Labs helped me understand its codebase and whenever I got stuck mentors were readily available for help.

Other than contributions, Playing with Music Blocks is fun. Before this, I had not experienced such an environment where we could learn concepts of music and programming and play altogether. I have also done a project in Music Blocks named

Elephant Ring Song. Other than that I have also set up and explored the lesson plan generator and created some lesson plans like Wonderwall Magic with Brown Eyed Girl.

But while generating the lesson plans I've observed that

1. The current system lacks the ability to create code and Musicblocks' project-specific data (I've added some examples and Demos ahead).
2. The current implementation is tightly coupled with one particular model (llama3) which will make it hard to adapt to newer and more advanced models in the future.

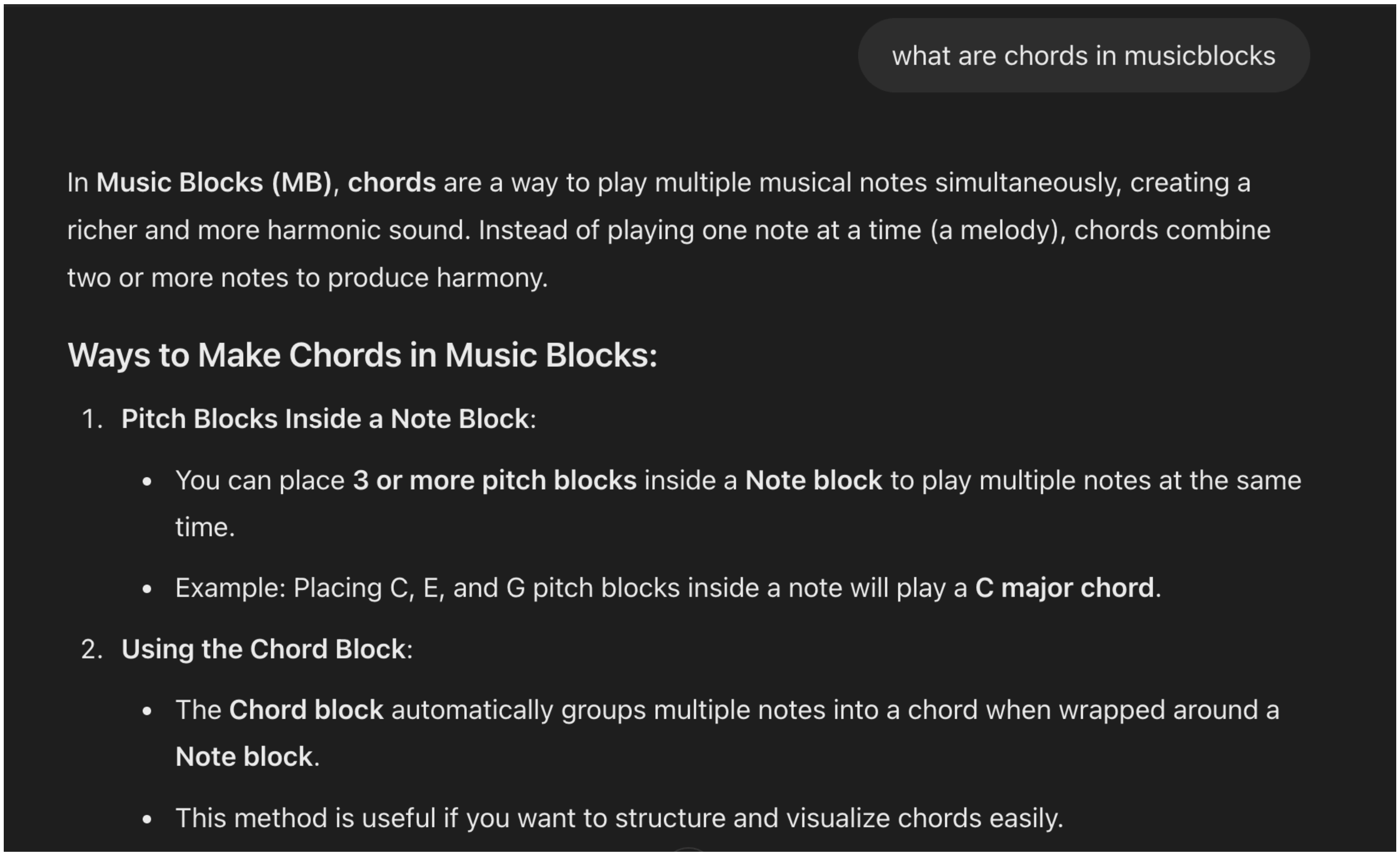
It will be a lot of fun if it also generates the code snippets for the Musicblocks projects and has a model abstraction layer to swap the models easily in the future.

So, Following this, I want to use my familiarity with the codebase and love to play with Music Blocks to add **AI Code generation for lesson plans and model abstraction layer**.

This project will impact Music Blocks in the following ways:

1. **Automated Code Generation:** Adding AI-driven code generation for lesson plans means that the system can automatically generate project-specific code snippets. This saves time and reduces manual work, allowing educators and students to focus on content and learning effectively.

   Example: Understanding Chords in Music Blocks.

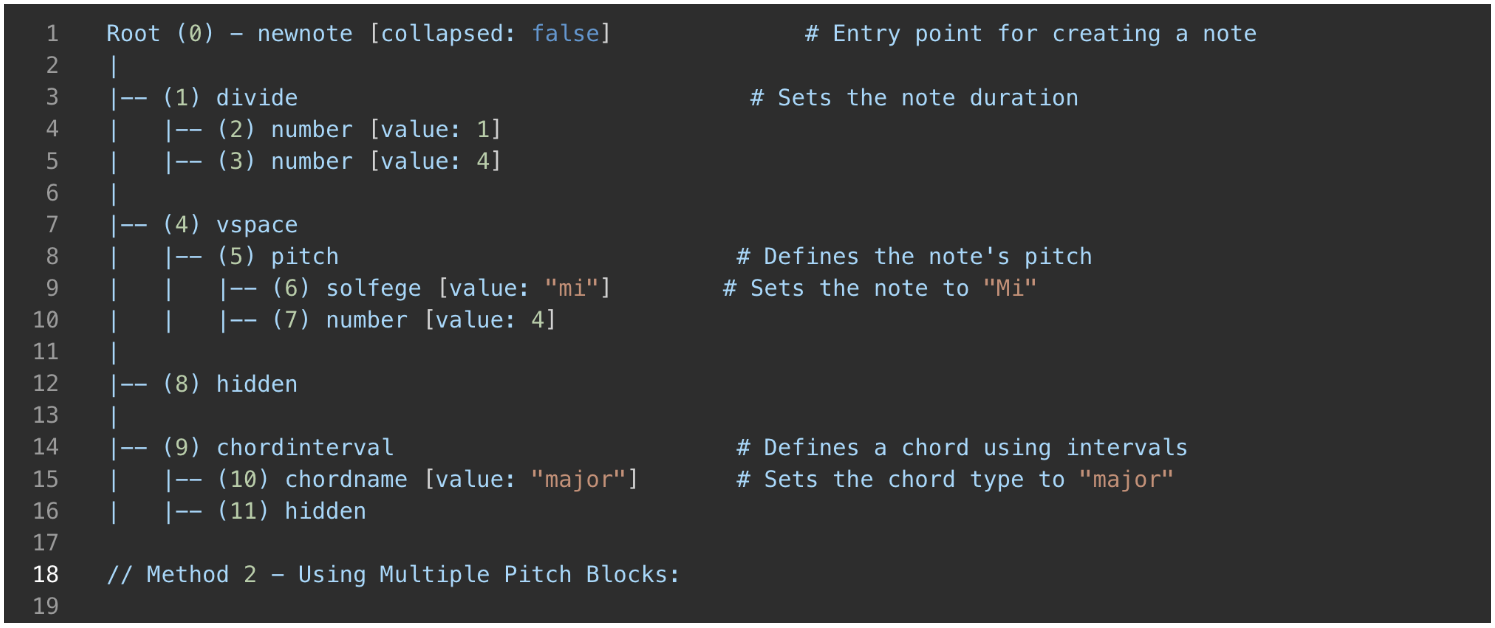


what are chords in musicblocks

In **Music Blocks (MB)**, **chords** are a way to play multiple musical notes simultaneously, creating a richer and more harmonic sound. Instead of playing one note at a time (a melody), chords combine two or more notes to produce harmony.

Ways to Make Chords in Music Blocks:

1. **Pitch Blocks Inside a Note Block**:
   - You can place **3 or more pitch blocks** inside a **Note block** to play multiple notes at the same time.
   - Example: Placing C, E, and G pitch blocks inside a note will play a **C major chord**.
2. **Using the Chord Block**:
   - The **Chord block** automatically groups multiple notes into a chord when wrapped around a **Note block**.
   - This method is useful if you want to structure and visualize chords easily.



```
Root (0) – newnote [collapsed: false]             # Entry point for creating a note
|
|-- (1) divide                                   # Sets the note duration
|    |-- (2) number [value: 1]
|    |-- (3) number [value: 4]
|
|-- (4) vspace
|    |-- (5) pitch                               # Defines the note's pitch
|    |    |-- (6) solfege [value: "mi"]         # Sets the note to "Mi"
|    |    |-- (7) number [value: 4]
|
|-- (8) hidden
|
|-- (9) chordinterval                            # Defines a chord using intervals
|    |-- (10) chordname [value: "major"]         # Sets the chord type to "major"
|    |-- (11) hidden

// Method 2 – Using Multiple Pitch Blocks:

```

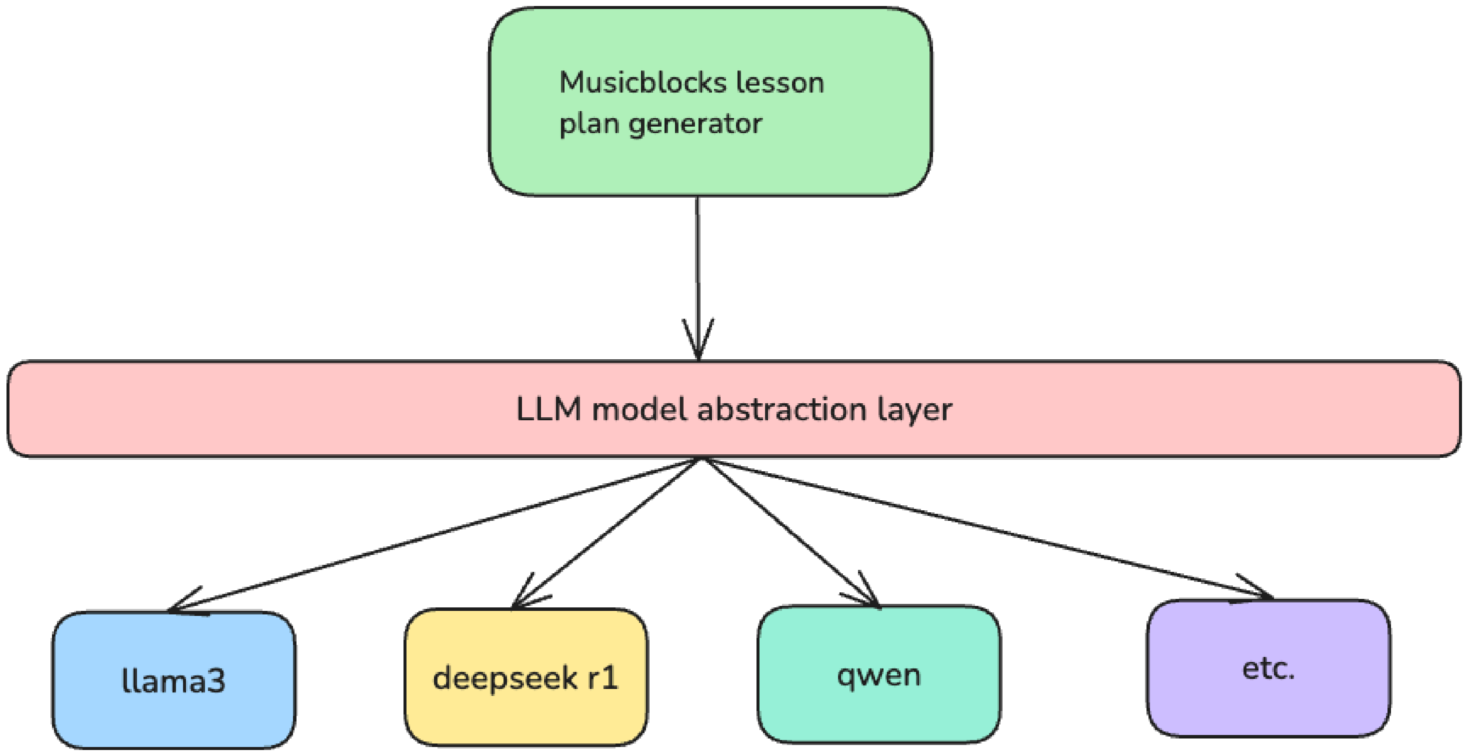
```
// Method 2 - Using Multiple Pitch Blocks:

Root (0) - newnote [collapsed: false]              # Entry point for creating a note
|
|-- (1) divide                                 # Sets the note duration
|   |-- (2) number [value: 1]
|   |-- (3) number [value: 4]
|
|-- (4) vspace
|   |-- (5) pitch                              # Defines the first pitch
|   |   |-- (6) solfege [value: "mi"]        # Sets the note to "Mi"
|   |   |-- (7) number [value: 4]
|   |
|   |-- (9) pitch                              # Defines the second pitch
|   |   |-- (10) solfege [value: "so"]       # Sets the note to "Sol"
|   |   |-- (11) number [value: 4]
|   |
|   |-- (12) pitch                             # Defines the third pitch
|   |   |-- (13) solfege [value: "re"]       # Sets the note to "re"
|   |   |-- (14) number [value: 4]
|
|-- (8) hidden

```

2. **Improved Customization and Relevance:** With the ability to generate Music Blocks–specific data, lesson plans can be tailored more closely to the project's context. This results in more personalized and context-aware content, enhancing the overall learning experience.

   A good example of this can be seen here.

3. **Enhanced Flexibility:** By introducing a model abstraction layer, the system becomes independent of a single model (llama3). This makes it far easier to integrate newer, more advanced models as they become available, ensuring the platform remains cutting-edge over time.

   Workflow:


Musicblocks lesson plan generator
LLM model abstraction layer
llama3
deepseek r1
qwen
etc.

4. **Enhanced User Engagement:** Dynamically generated code snippets add an element of interactivity and creativity, which can make the learning process more engaging and enjoyable for users.

Overall, these enhancements will lead to a more context-aware, versatile, and user-friendly lesson plan generator system—positioning Music Blocks as a forward-thinking tool in educational technology. It can help enhance the effectiveness of the Sugar Learning Platform and contribute to a more engaging and interactive learning experience for teachers and students worldwide.

Prior Experience and Skills

I've been doing software engineering for the past three years, and use JavaScript, TypeScript, Python, React, Tailwind, LLMs, SAM model, and other Generative AI libraries and frameworks. For the past 1 to 1.5 years, I've been contributing to Sugar Labs as a contributor in Music Blocks. Some of my contributions are mentioned below.

PR Link	Description	Status
#3885	Added a feature to delete the blocks in bulk effectively.	**Merged**
#3912	Implemented collaboration space for the real-time collaboration.	**Merged**
#3923	Installed and configured socket.io library for the collaboration.	**Merged**
#3938	Implemented the logic to make a socket connection from the Music Blocks's client to the collaboration server.	**Merged**
#3952	Built the functionality to send the changes of adding blocks in the MB project.	**Merged**
#3960	Added the functionality to send the changes of move, connect, disconnect, value-update, and delete events.	**Merged**
#3963	Implemented the mechanism to create the room for the collaboration.	**Merged**
#3966	Added the link creation mechanism to share to invite the peers.	**Merged**
#3984	Created the collaboration mouse along with custom and random names.	**Merged**
#3967	Fixed the multi-mouse bug on block creation and collaboration link text.	**Merged**

#4014	Ideated and executed edge case of exiting the user from the collaboration on `New Project` and `Load project from file` events.	**Merged**
#4034	Wrote the comprehensive documentation for collaboration functionality.	**Merged**
#Link	Engineered a scalable Express.js backend server, using Socket.IO for data synchronization with room management and server-side logging.	**Merged**
#4146	Fixed the `jest` dependency issue and removed extra configs from `jest.config.js`.	**Merged**
#4157	Solved the issues of deprecated dependencies and the old version package-lock.json.	**Merged**
#3468	Fixed the position of the trashcan.	**Merged**
#3461	Implemented the tooltips in the JavaScript editor.	**Merged**
#3507	Fixed duplicate pitches issue in the phrase maker.	**Merged**
#3731	Added the ctrl+z shortcut for the undo feature for better accessibility.	**Merged**
#4225	Used the nginx to proxy browser requests to collaboration-server service in docker.	**In review**
#4331	Fixed slow-loading issues of Music Blocks on all the browsers.	**Merged**

I have raised 50+ PRs and a complete list of my PRs can be found here. Also, I have opened 25+ issues and fixed them which can be found here.

Academic Experience and Projects

As part of my Academic learning, I have made the following projects Projects.

- **Opensox.in** - This app lets users find open-source projects blazingly fast. Since its launch, it got 79,000+ impressions, 33,000+ visitors, 12,600+ queries, 1,500+ bookmarks, 550+ discord community members, 500+ signups, and 50+ countries. It is built using Typescript, NextJS, Postgresql, NextAauth, ExpressJS, and Zustand and deployed on Vercel and Railway.

- **drawRTC** - drawRTC enables users to collaborate on drawings and edit the text in real-time with friends. The application includes tools for drawing, a real-time text editor, and the ability to save creations as PNG or PDF. It is built using Node.js for the backend, React for the frontend, and WebSockets for seamless interaction. Docker can be used for easy setup.

- **editorX** – editorX is an advanced image editing tool that allows users to interactively modify images with precision. Users can click on any area of an image for automatic segmentation and apply inpainting to transform that area using custom prompts. The application includes powerful editing features such as background removal, image rotation, flipping, resizing, and eight customizable filters with adjustable intensity. Users can also reset edits, switch between dark and light modes, and download the final image. It is built using the **FLUX API** for inpainting and the **Segment Anything Model** for accurate image segmentation, providing a seamless and responsive editing experience.

Project Size

I am applying for a large project (~300-350 hours).

Project Timeframe

8 May 2024 to 1 September 2024

Contact info and timezone(s)

Primary Email: ajeetpratap517@gmail.com

Secondary Email: ajeetpratapsingh351@gmail.com

Github: **@apsinghdev**

Matrix Id: **@ajeit2023:matrix.org**

Discord ID: **apsinghdev**

Language: Hindi (Native), English (Fluent)

Location: Kanpur, India

Time Zone: IST (GMT+5:30)

Preferred mode of Communication: Email, Google Meet, Jitsi, Matrix

Time Commitment

I am on summer vacation from 5th May to 10th July. In this time frame, I would be able to devote **~40-45 hrs/week**. After that, I would be able to devote **~20-30 hrs/week.** which may increase if the need arises. I am working on the GSoC project from around 8th May to 1st September timeframe (**Note:** can be extended if the need arises).

S. No	Dates	Days (Total)	Time Commitment

1.	8th May - 10th July	Mon-Sun (7)	8 hrs/day (Mon-Sun)
2.	10th July - 1st Sep	Mon-Sun (7)	6 hrs/day (Mon-Sun)

Estimated Total Working Days: 90-100

Estimated Hours: 350-400 hours (This may change as per requirements).

To report my progress, I will provide detailed progress updates by writing weekly blogs, outlining the tasks completed, challenges faced, and plans for the upcoming week. These updates will be shared on the project's mailing list or designated communication channel.

Technical Requirements

As per the discussions in the meetings and on the Music Blocks's Matrix (Element) server, we'd need the following technologies for this project:

1. An Open source LLM
2. A vector database
3. FastAPI
4. Approximate Nearest Neighbor (ANN) algorithms

Benefits of using an Open-source LLM

- **Customizability:** An open-source LLM gives us full control over the implementation of fine-tuning and RAG, allowing us to train it specifically on Music Blocks (MB) data for better lesson plan generation.
- **Transparency:** Since the model is open-source, we can inspect and modify it as

needed, ensuring transparency and avoiding reliance on proprietary solutions.

- **Cost-effectiveness:** Using an open-source model reduces costs significantly compared to paid APIs, making it a cost-effective choice for long-term scalability.
- **Flexibility:** We can experiment with different models and architectures to optimize performance for our specific use case.

Benefits of using a Vector Database

- **Efficient Retrieval:** A vector database stores lesson plans and MB-related content as embeddings, making retrieval more efficient and contextually relevant.
- **Enhanced RAG:** By integrating a vector database, we can improve retrieval-augmented generation (RAG), ensuring the LLM fetches more accurate and relevant information before responding.
- **Scalability:** It scales efficiently, allowing us to manage a growing dataset while maintaining fast query times.
- **Better Context Handling:** A vector database allows us to store and retrieve relevant past interactions, reducing the chances of context loss during conversations.

Benefits of using FastAPI

- **High Performance:** FastAPI provides an asynchronous, high-performance API that ensures quick responses, making the system more efficient.
- **Seamless Deployment:** It simplifies the deployment of our lesson plan generator, allowing the community to test and provide feedback.
- **Automatic Documentation:** FastAPI generates OpenAPI documentation automatically, making it easier for developers to integrate and maintain the system.

- **Type Safety and Validation:** FastAPI leverages Python's type hints and Pydantic for automatic data validation and serialization. This ensures that the inputs and outputs of your API are always validated, reducing errors and improving the reliability of our lesson plan generator.

Benefits of using Approximate Nearest Neighbor (ANN) Algorithms

- **Faster Searches:** ANN algorithms speed up similarity searches, helping users quickly find relevant lesson plans based on their queries.
- **Improved Context Awareness:** By leveraging ANN, we can improve context awareness, ensuring the model fetches past interactions and related topics effectively.
- **Optimized for Large Datasets:** These algorithms are optimized for handling large datasets, making them ideal for your lesson plan generator.
- **Low Latency:** ANN-based search techniques provide real-time retrieval, ensuring minimal delay in fetching relevant lesson plans and content.

These are the proposed technologies required for the project. As I continue discussions with the mentors, I will refine the requirements and make adjustments as needed. The final selection of technologies will be determined based on these discussions and project needs.

Other Summer Obligations,

I have no commitments in the summer. I'll be staying back home for most of it. I have mentioned my typical working hours above and on average will be able to spend 45-50 hours per week on the project.

Communication Channels

I am active on Emails and the Matrix (Element) app. I can work with whatever platform my mentor prefers. Meetings can be held every week to discuss progress in the project.

Section 2: Proposal Details

Problem Statement

Project	**AI Code generation for lesson plans and model abstraction layer**
Target Audience	● Music Blocks Users, Teachers, and learners around the world.
Core User Need	● **AI Code generation**: Adding the MB-related projects to the database and updating its RAG prompts to adapt to the MB code would improve the response of the project efficiently. It will also provide code snippets/starter code to build projects more intuitively. ● **LLM model abstraction layer**: A model abstraction layer will improve the system effectively as it'll be way easier to do the testing with different models and swap the models when we have a better one.

Section 2.1: WHAT (**Key Milestones**)

1. **Create a baseline to benchmark the RAG system**
    a. Design Comprehensive Evaluation Framework Before Implementation
    b. Make initial setup to retrieve docs using BM25
    c. Creation of a baseline using BM25 to benchmark the RAG system
    d. Collect a test dataset related to lesson plans and MB project code
    e. Retrieve Results from Both Systems (BM25 and Vector Search)
    f. Evaluate Performance Using Key Metrics
2. **Generate Music Blocks project code with better context.**
    a. Improve the database by adding more Music Blocks (MB) project data.
    b. Update RAG prompts to integrate MB-specific code efficiently.
    c. Provide better starter code snippets to guide project creation.
3. **Show the code as a nice snippet in the conversation.**
    a. Format the generated code for better readability.
    b. Use syntax highlighting and proper indentation.
    c. Ensure snippets include explanations where necessary.
4. **Generate the related comments for better understanding.**
    a. Add contextual comments to explain key parts of the generated code.
    b. Include references to MB concepts where relevant.
    c. Ensure comments align with the learning goals of the lesson plan.
5. **Make the generated code shareable (ie. pasting in Music Blocks).**
    a. Ensure the generated code can be directly copied and used in MB.
    b. Format outputs to align with MB's expected code structure.
    c. Provide a button or mechanism for easy sharing/exporting.
6. **Implement the model abstraction layer in the current RAG system.**
    a. Decouple the system from a specific model to enable easy switching.
    b. Make the system adaptable to newer models like DeepSeek R1.
    c. Structure the architecture for efficient model experimentation.
7. **Make response retrieval faster with ANNs.**
    a. Optimize retrieval mechanisms using ANNs.

   b. Improve indexing and search efficiency within the RAG system.
   c. Reduce response latency while maintaining context accuracy.

8. **Minimize the hallucination**
   a. Implement techniques like Query Expansion & Reformulation.
   b. Introduce Context Filtering and a Better Chunking Strategy.
   c. Test and refine hallucination-reduction techniques based on performance.

Section 2.2: HOW

I have divided this project into three parts.

1. **Creation of a baseline using BM25 to benchmark the RAG system**
2. **Implementation of AI code generation for lesson plans**
3. **Implementation of Model Abstraction Layer in the RAG System.**
4. **Optimization of Response Retrieval and Minimization of Hallucination.**

Part 1: Creation of a baseline using BM25 to benchmark the RAG system.

As discussed with mentors, It's really important to create a baseline using BM25 to benchmark the RAG system. Here are the implementation steps to create a baseline:

- **Design Comprehensive Evaluation Framework Before Implementation**

  We will develop an unbiased evaluation framework before implementing either

BM25 or vector search systems for our RAG solution. This framework will include categorized test queries spanning factual, procedural, and troubleshooting scenarios relevant to Music Blocks, along with manually annotated gold-standard answers for each query. By establishing clear metrics (Precision@K, Recall@K, nDCG, MRR) and creating automated evaluation scripts upfront, we ensure our implementation decisions are guided by objective performance criteria rather than being tailored to specific test cases. This approach will facilitate fair comparison between retrieval methods, provide consistent benchmarking throughout development, and ultimately lead to a more robust and generalizable RAG system for Music Blocks.

- **Make initial setup to retrieve docs using BM25**

  We will implement a BM25-based document retrieval system to establish a performance baseline for our RAG model. This setup will preprocess and index documents, enabling efficient term-based matching. By integrating BM25 alongside the existing vector search, we can compare their effectiveness in retrieving relevant content. This dual-system approach allows us to evaluate and optimize our RAG model's accuracy, ensuring more comprehensive and reliable information retrieval. Here is a simple overview of its implementation:

```python
from rank_bm25 import BM25Okapi
from nltk.tokenize import word_tokenize

# Sample documents
documents = [#sample documents]

# Tokenize the documents
tokenized_corpus = [word_tokenize(doc.lower()) for doc in documents]
```

```python
# Initialize the BM25 model
bm25 = BM25Okapi(tokenized_corpus)

# Example user query
query = "chords"
tokenized_query = word_tokenize(query.lower())

# Retrieve top-k documents (e.g., top 3)
top_docs = bm25.get_top_n(tokenized_query, documents, n=3)

# Display results
print("Query:", query)
print("Top 3 Documents:")
for i, doc in enumerate(top_docs, 1):
    print(f"{i}. {doc}")
```

- **Collect a test dataset related to lesson plans and MB project code**

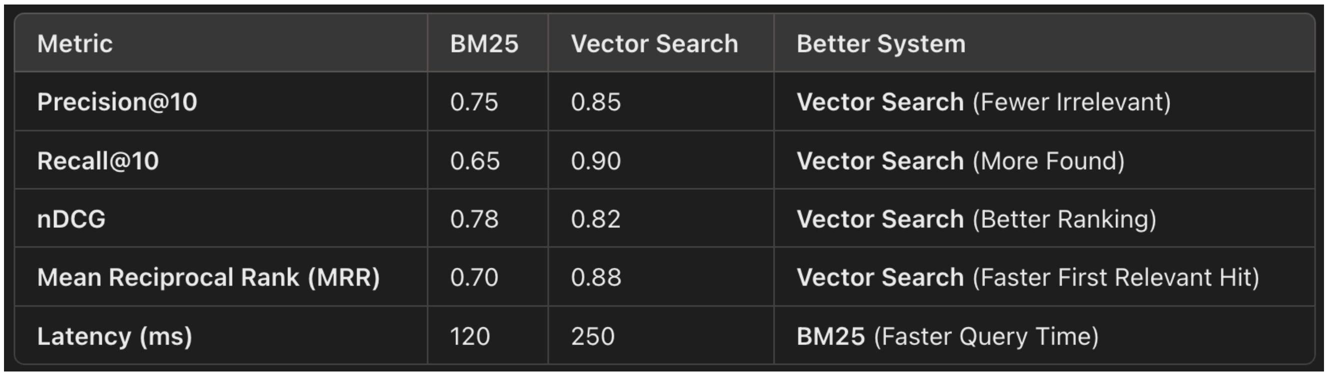
  We will gather a comprehensive dataset comprising lesson plans and Music Blocks (MB) project code to evaluate the RAG system's performance. This dataset will include diverse examples of instructional content, user-generated MB projects, and code snippets. By curating data from real-world use cases, we ensure the benchmarking process reflects authentic query patterns. This dataset will serve as a reliable reference point to compare BM25 and vector search, facilitating accurate performance assessment and system optimization.

- **Retrieve Results from Both Systems (BM25 and Vector Search)**

  We will retrieve and compare results from both the BM25 and vector search systems using the same set of queries. For each query, we will collect the top-ranked documents from both approaches. This parallel retrieval allows us to analyze how each system performs in terms of relevance and accuracy. By systematically capturing outputs from both methods, we can identify strengths, weaknesses, and areas for improvement, providing a solid foundation for benchmarking and optimizing the RAG system.

- **Evaluate Performance Using Key Metrics**

  Finally, we'll be able to assess the performance of BM25 and vector search using key evaluation metrics such as Precision@K, Recall@K, and nDCG (Normalized Discounted Cumulative Gain). These metrics will measure how accurately each system retrieves relevant documents, how many relevant documents are found, and how well the results are ranked. By systematically analyzing these performance indicators, we can determine which method delivers better results, identify areas for optimization, and ensure the RAG system meets the desired accuracy and efficiency standards. Example:



Metric	BM25	Vector Search	Better System
Precision@10	0.75	0.85	**Vector Search** (Fewer Irrelevant)
Recall@10	0.65	0.90	**Vector Search** (More Found)
nDCG	0.78	0.82	**Vector Search** (Better Ranking)
Mean Reciprocal Rank (MRR)	0.70	0.88	**Vector Search** (Faster First Relevant Hit)
Latency (ms)	120	250	**BM25** (Faster Query Time)

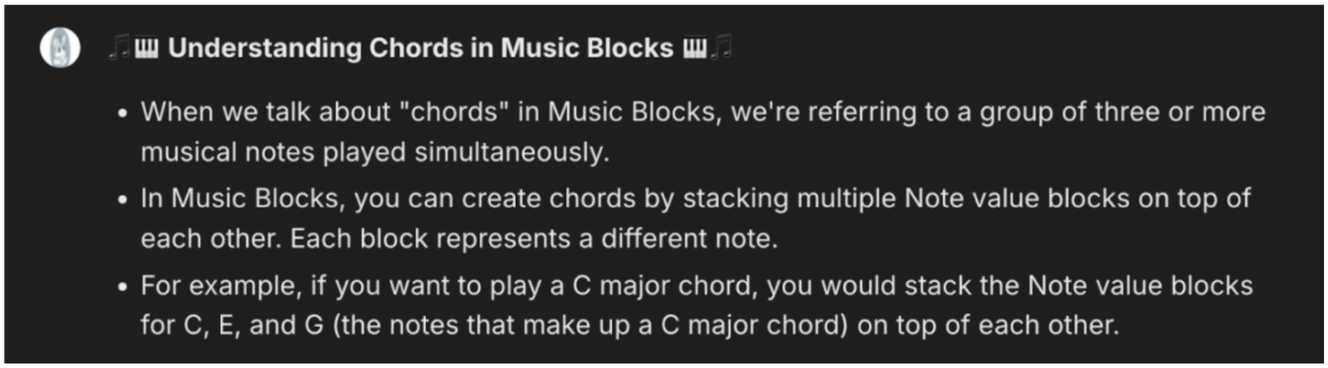
These evaluations and benchmarking will guide us to find weaknesses in our RAG system and will help us make the system more useful to the end user.

End-User Outcomes

The end-user outcomes of this implementation will be:

- Users will receive **more relevant, precise,** and **contextually correct** results as the evaluation helps identify which search method retrieves better information. Example:

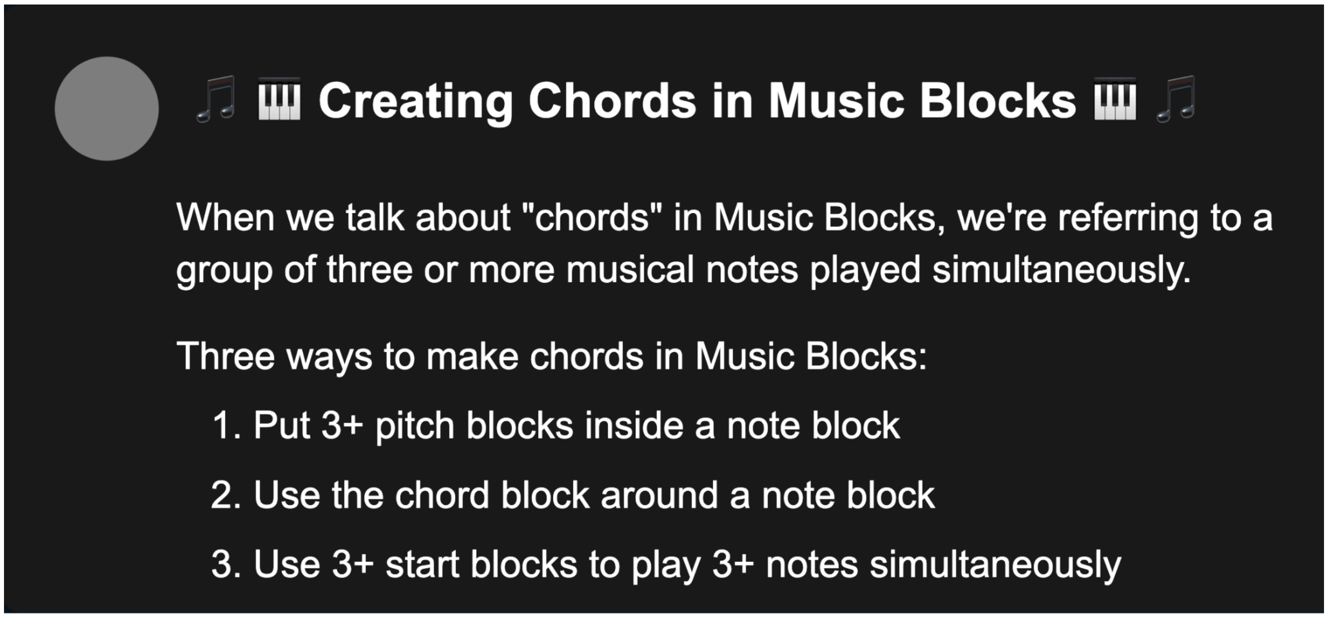
  (Before)



🎵🎹 **Understanding Chords in Music Blocks** 🎹🎵

- When we talk about "chords" in Music Blocks, we're referring to a group of three or more musical notes played simultaneously.
- In Music Blocks, you can create chords by stacking multiple Note value blocks on top of each other. Each block represents a different note.
- For example, if you want to play a C major chord, you would stack the Note value blocks for C, E, and G (the notes that make up a C major chord) on top of each other.

(After)



🎵 🎹 **Creating Chords in Music Blocks** 🎹 🎵

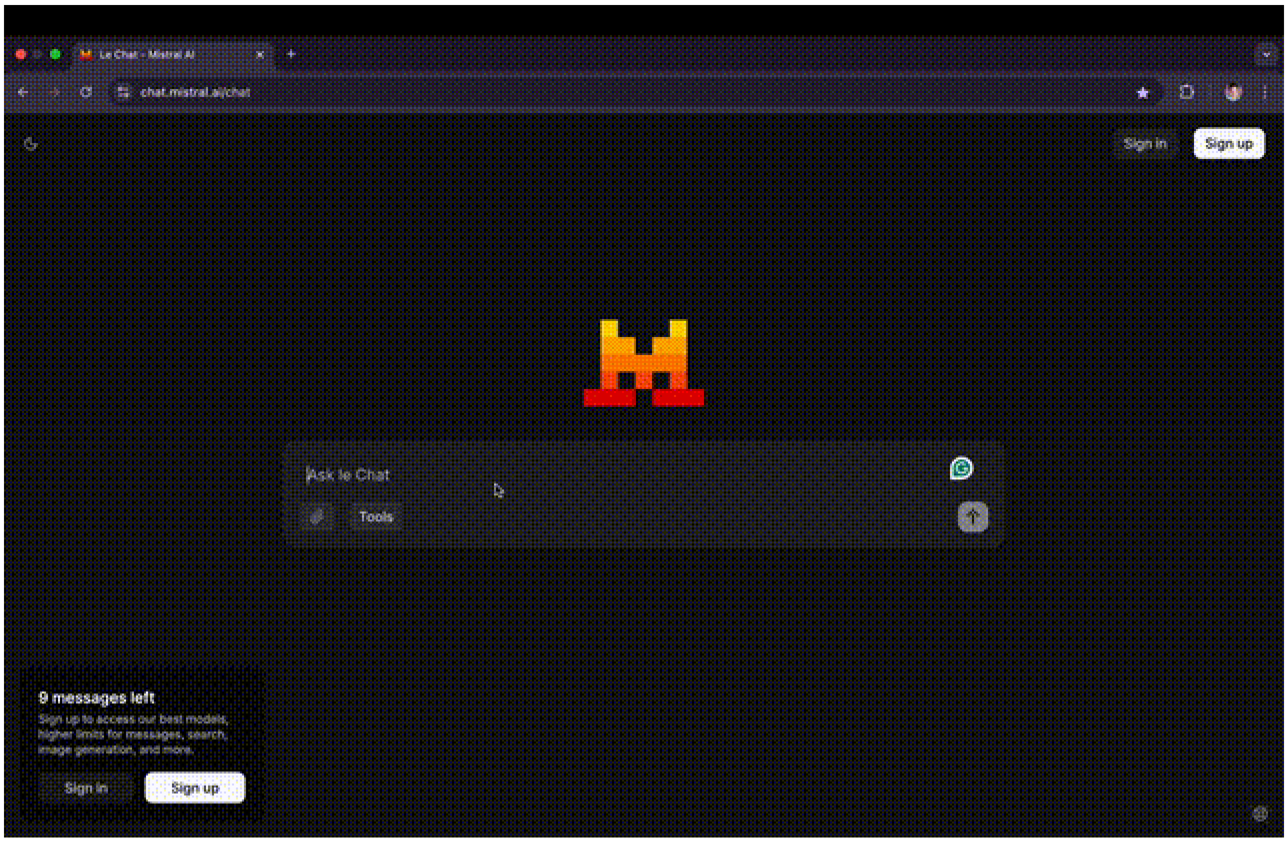
When we talk about "chords" in Music Blocks, we're referring to a group of three or more musical notes played simultaneously.

Three ways to make chords in Music Blocks:

1. Put 3+ pitch blocks inside a note block
2. Use the chord block around a note block
3. Use 3+ start blocks to play 3+ notes simultaneously

- Evaluating latency helps balance between accuracy and speed, ensuring quick query processing without compromising quality. Demo:

(Please click anywhere on the image to watch the demo)



In this demo, you can observe the smoothness and speed of the responses. This is what I have envisioned for the lesson plan generator.

Part 2: Implementation of AI code generation for lesson plans.

User Management and Collaboration Setup can be implemented by following these steps.

- **Improve the database by adding more Music Blocks (MB) project data.**

  We need to improve the database by adding more Music Blocks (MB) project data to the existing dataset. This data follows a structured format, representing musical elements like notes, timbre, pitch, and rhythm. By expanding the dataset, we ensure that the Retrieval-Augmented Generation (RAG) system gets the relevant context needed to generate accurate and meaningful code. This enhancement will help the system better understand musical constructs and interactions. With more comprehensive data, we can improve the precision and efficiency of MB-related code generation. This is the sample of data that we have to add:

```
[[0,["start",{"id":1741076077436,"collapsed":false,"xcor":0,"y
cor":0,"heading":0,"color":-10,"shade":60,"pensize":5,"grey":1
00}],763,100,[null,1,null]],[1,"settimbre",777,141,[0,2,4,3]],
[2,["voicename",{"value":"guitar"}],928,141,[1]],[3,"hidden",7
77,677,[1,null]],[4,["newnote",{"collapsed":false}],791,173,[1
,5,8,12]],[5,"divide",893,173,[4,6,7]],[6,["number",{"value":1
}],979,173,[5]],[7,["number",{"value":4}],979,205,[5]],[8,"vsp
ace",805,205,[4,9]],[9,"pitch",805,237,[8,10,11,null]],[10,["s
olfege",{"value":"sol"}],879,237,[9]],[11,["number",{"value":4
}],879,269,[9]],[12,"hidden",791,331,[4,13]],[13,["newnote",{"
collapsed":false}],791,331,[12,14,17,21]],[14,"divide",893,331
,[13,15,16]],[15,["number",{"value":1}],979,331,[14]],[16,["nu
mber",{"value":4}],979,363,[14]],[17,"vspace",805,363,[13,18]]
,[18,"pitch",805,395,[17,19,20,null]],[19,["solfege",{"value":
"mi"}],879,395,[18]],[20,["number",{"value":4}],879,427,[18]],
[21,"hidden",791,489,[13,22]],[22,["newnote",{"collapsed":fals
e}],791,489,[21,23,26,30]],[23,"divide",893,489,[22,24,25]],[2
4,["number",{"value":1}],979,489,[23]],[25,["number",{"value":
```

```
2}],979,521,[23]],[26,"vspace",805,521,[22,27]],[27,"pitch",80
5,553,[26,28,29,null]],[28,["solfege",{"value":"sol"}],879,553
,[27]],[29,["number",{"value":4}],879,585,[27]],[30,"hidden",7
91,647,[22,null]]]
```

- **Update RAG prompts to integrate MB-specific code efficiently.**

  We’ll update the RAG prompts to efficiently integrate MB-specific code into the system. By refining the contextualization process, we ensure that user queries related to Music Blocks are accurately reformulated while preserving their intent. The updated prompts will prioritize MB-related terms, helping the system generate more relevant responses and improving code generation. This enhancement will optimize how the RAG model interprets and structures queries, making interactions more precise and aligned with Music Blocks' functionalities.

  The code snippet below is the overview of this functionality.

```
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain.llms import LlamaCpp
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings

vectorstore = FAISS.load_local("music_blocks_db",
HuggingFaceEmbeddings())

mb_prompt_template = """
You are an AI assistant specializing in Music Blocks. Reformulate the
user query to enhance retrieval accuracy by prioritizing MB-related
terms and MB code. Ensure the intent remains the same while
emphasizing MB concepts and MB code.
```

```
Original Query: {query}
Reformulated Query:
"""

prompt = PromptTemplate(template=mb_prompt_template,
input_variables=["query"])

def retrieve_mb_documents(user_query):
    reformulated_query = prompt.format(query=user_query)
    docs = vectorstore.similarity_search(reformulated_query, k=5)
    return docs

llm = LlamaCpp(
    model_path="path_to_your_llama_model.bin",
    n_ctx=2048,
    n_gpu_layers=1,
    temperature=0.7,
)

mb_chain = LLMChain(llm=llm, prompt=PromptTemplate(
    template="Given the following MB-related context, generate a
response including MB code if applicable:\n\n{context}\n\nUser Query:
{query}\nAI Response:",
    input_variables=["context", "query"]
))

def mb_rag_pipeline(user_query):
    relevant_docs = retrieve_mb_documents(user_query)
    context = "\n".join([doc.page_content for doc in relevant_docs])
    response = mb_chain.run(context=context, query=user_query)
    return response

user_input = "How do I create a new action block in Music Blocks using
MB code?"
response = mb_rag_pipeline(user_input)
print(response)
```

- **Provide better starter code snippets to guide project creation.**

  To enhance project creation, we'll provide structured starter code snippets that represent hierarchical data in a tree-like format. These snippets will serve as a foundation for building the Music Blocks project and will give additional context to the beginners.

  The code snippets generated by the RAG system will be similar to this:

```
Root (0) - start [id: 1741076077436]
|
|-- (1) settimbre
|   |-- (2) voicename [value: "guitar"]
|   |-- (3) hidden
|
|-- (4) newnote
|   |-- (5) divide
|   |   |-- (6) number [value: 1]
|   |   |-- (7) number [value: 4]
|   |
|   |-- (8) vspace
|   |   |-- (9) pitch
|   |   |   |-- (10) solfege [value: "sol"]
|   |   |   |-- (11) number [value: 4]
|   |
|   |-- (12) hidden
```

  Note: I am researching some better ways to present the code snippets so that they are simple enough for kids and teachers to understand and effective enough to provide solid learning.

- **Improve code readability with syntax highlighting, indentation, comments, and explanations.**

  We can improve code readability by using syntax highlighting to distinguish keywords and variables making the code easier to follow. Also, adding meaningful comments will allow us to explain complex sections, making it easier for others (and our future selves) to understand. By providing concise explanations and documentation, we ensure that our code remains accessible and easy to use.

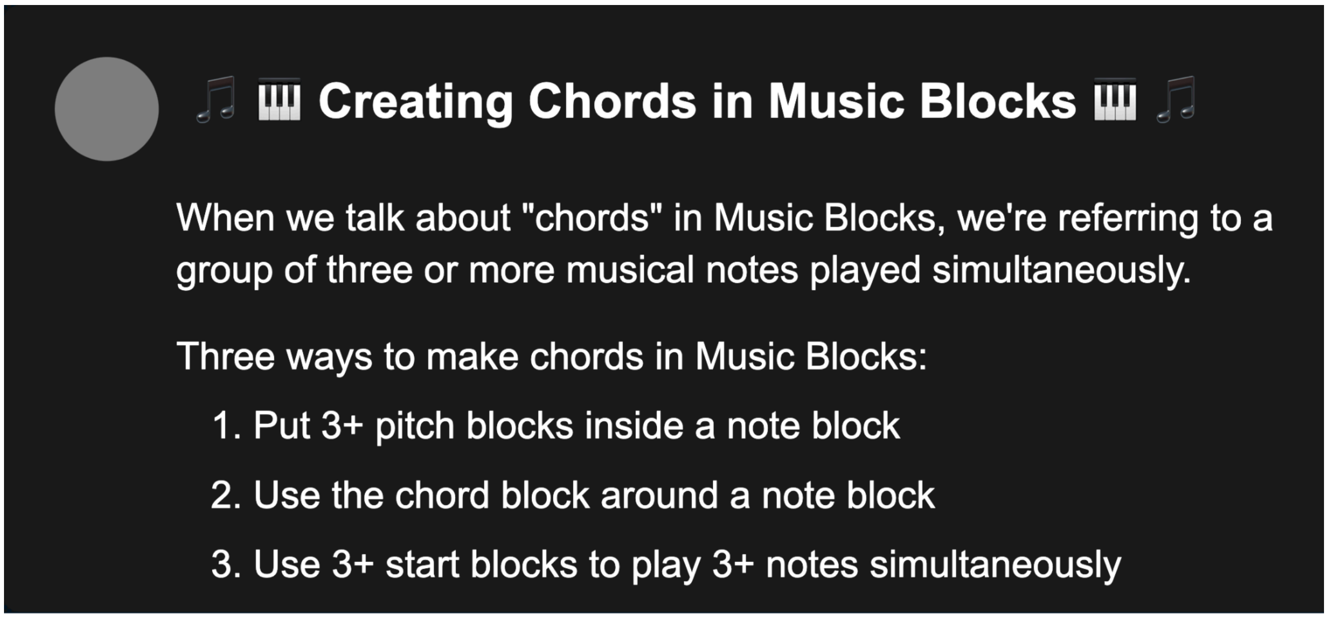
```
Root (0) - start [id: 1741076077436]      # Entry point of the Project
|
|-- (1) settimbre                         # Sets the instrument sound
|   |-- (2) voicename [value: "guitar"]   # Specifies the instrument
|   |-- (3) hidden
|
|-- (4) newnote                           # Creates a new musical note
|   |-- (5) divide                        # Sets the note duration
|   |   |-- (6) number [value: 1]
|   |   |-- (7) number [value: 4]
|   |
|   |-- (8) vspace                        # Vertical spacing block
|   |   |-- (9) pitch                     # Defines the note's pitch
|   |   |   |-- (10) solfege [value: "sol"]   # Sets the note to "Sol"
|   |   |   |-- (11) number [value: 4]    # Sets the octave
|   |
|   |-- (12) hidden
```

  Note: If I find a better way to show the code along with comments, I'll update the above code snippet to a better format.

- **Include references to MB concepts where relevant.**

We will include references to Music Blocks (MB) concepts wherever relevant to make our lesson plans more informative and contextually accurate. By directly linking MB concepts, we ensure that the generated content aligns with the project's core ideas and functionalities. This will help learners better understand how MB principles apply in real-world scenarios. Providing these references will also enhance the credibility of our lesson plans and reduce confusion.

Something similar to this but with more specific context and details:



🎵 🎹 **Creating Chords in Music Blocks** 🎹 🎵

When we talk about "chords" in Music Blocks, we're referring to a group of three or more musical notes played simultaneously.

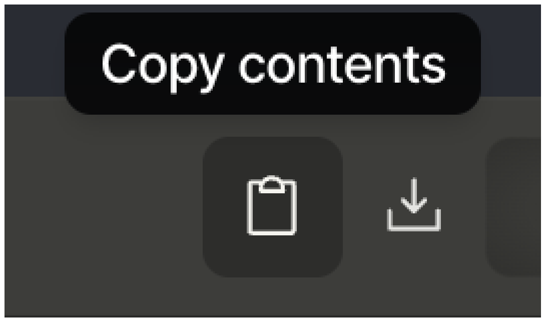
Three ways to make chords in Music Blocks:

1. Put 3+ pitch blocks inside a note block
2. Use the chord block around a note block
3. Use 3+ start blocks to play 3+ notes simultaneously

- **Make code easily copyable and usable in MB with export options.**

We can make code easily copyable and usable in Music Blocks by providing export options. This will allow users to quickly integrate generated code into their MB projects without manual adjustments. By offering well-formatted, ready-to-use code snippets, we can ensure a seamless experience for learners and educators. Export options will also enhance flexibility, enabling users to save and reuse code efficiently in different lesson plans and projects.

For the implementation of this, we can add two buttons, **Copy** and **Download**, like this:



Part 3: Implementation of Model Abstraction Layer in the RAG System.

After part 1, I will test the code generation and will start implementing the Model Abstraction Layer in the system in the following steps:

- **Define the Abstraction Layer for the system.**

  The abstraction layer will provide a consistent interface for interacting with different models (e.g., retrieval models, and language models). This involves defining abstract classes or interfaces for retrieval and generation tasks.

  Example:

```python
from abc import ABC, abstractmethod

class RetrievalModel(ABC):
    @abstractmethod
    def retrieve(self, query: str, top_k: int) -> list:
        """Retrieve relevant documents for a given query."""
        pass

class GenerationModel(ABC):
    @abstractmethod
    def generate(self, context: str, query: str) -> str:
        """Generate a response based on the context and query."""
        pass
```

- **Implement Concrete Models**

  After that, I'll create a concrete implementation of the abstract classes for specific models. For example, we might use a dense retriever like FAISS or a sparse retriever like BM25, and a language model like Llama or DeepSeek R1.

  Example:

```python
class FAISSRetriever(RetrievalModel):
    def __init__(self, index, model):
        self.index = index
        self.model = model

    def retrieve(self, query: str, top_k: int) -> list:
        query_embedding = self.model.encode(query)
        scores, indices = self.index.search(query_embedding, top_k)
        return indices

class GPTGenerator(GenerationModel):
    def __init__(self, model):
        self.model = model
    def generate(self, context: str, query: str) -> str:
        input_text = f"Context: {context}\nQuery: {query}\nAnswer:"
        return self.model.generate(input_text)
```

- **Integrate the Abstraction Layer into the RAG System**

  The RAG system will use the abstraction layer to interact with the retrieval and generation models. This ensures that the system is not tightly coupled to specific implementations.

  Example:

```
class RAGSystem:
    def __init__(self, retriever: RetrievalModel, generator:
GenerationModel):
        self.retriever = retriever
        self.generator = generator

    def answer(self, query: str, top_k: int = 5) -> str:
        # Retrieve relevant documents
        retrieved_docs = self.retriever.retrieve(query, top_k)

        # Combine documents into a single context
        context = " ".join(retrieved_docs)

        # Generate the final answer
        return self.generator.generate(context, query)
```

- **Configure and Use the System.**

After the successful implementation of the above steps, we can configure the RAG system with different models and use it to answer queries.

Example:

```python
# Initialize models
retriever = FAISSRetriever(index=faiss_index, model=embedding_model)
generator = GPTGenerator(model=llama_model)

# Create RAG system
rag_system = RAGSystem(retriever, generator)

# Answer a query
response = rag_system.answer("What is the note block in Music Blocks?")
print(response)
```

- **Extension and Customization**

  After this, we'll have more than enough room to extend and customize our RAG system. Here are a couple of ways we can do this.

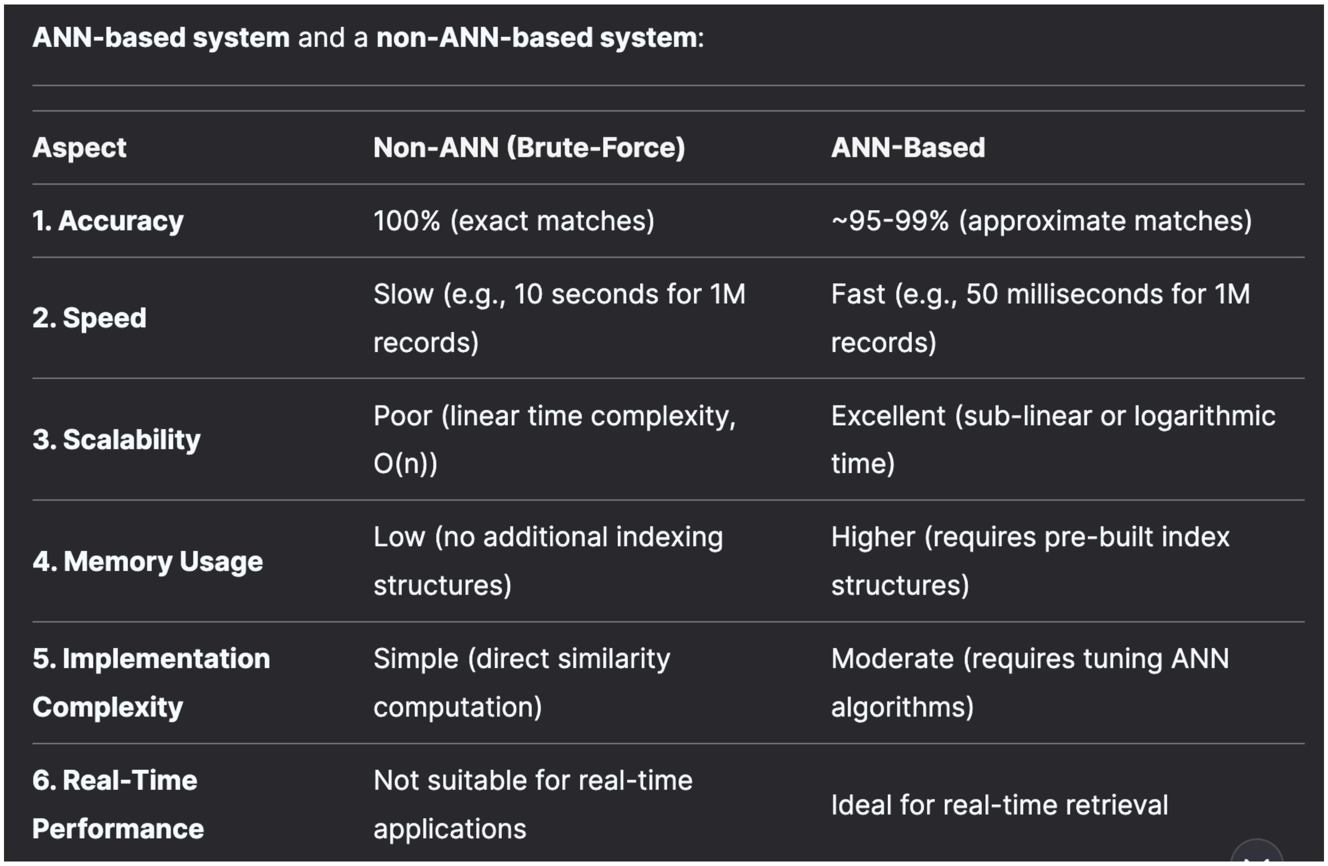
  1. To add a new retrieval or generation model, we can simply implement the corresponding abstract class.
  2. We can use configuration files (e.g., YAML, JSON) to manage model settings and switch between models dynamically.
  3. We can add error handling and logging to make the system robust.
  4. We can optimize the abstraction layer for performance if needed.

Part 4: Optimization of Response Retrieval and Minimization of Hallucination.

- **Optimize retrieval mechanisms using ANNs.**

  I'll also work on optimizing the retrieval mechanisms by using **Approximate Nearest Neighbors (ANN)** algorithms to improve the speed and accuracy of fetching relevant data. This approach allows us to efficiently search and retrieve information from large datasets by approximating the closest matches to a query. By implementing ANN-based retrieval, we will enhance the system's performance, reduce latency, and provide more precise and contextually relevant lesson plans. This optimization will also support scalability, ensuring the system remains responsive as data grows.

  Here is a comparison between the method we use currently and ANNs:


ANN-based system and a **non-ANN-based system**:

Aspect	**Non-ANN (Brute-Force)**	**ANN-Based**
1. Accuracy	100% (exact matches)	~95-99% (approximate matches)
2. Speed	Slow (e.g., 10 seconds for 1M records)	Fast (e.g., 50 milliseconds for 1M records)
3. Scalability	Poor (linear time complexity, O(n))	Excellent (sub-linear or logarithmic time)
4. Memory Usage	Low (no additional indexing structures)	Higher (requires pre-built index structures)
5. Implementation Complexity	Simple (direct similarity computation)	Moderate (requires tuning ANN algorithms)
6. Real-Time Performance	Not suitable for real-time applications	Ideal for real-time retrieval

Note: As discussed with the mentor here, the implementation of the ANNs is subjective and after implementing the code generation feature, I'll test if the need for ANNs arises and only then move forward with implementing it.

- **Improve indexing and search efficiency with techniques like Query Expansion & Reformulation.**

As part of the optimization process, I'll improve **indexing and search efficiency** within the **Retrieval-Augmented Generation (RAG)** system by optimizing data structures and implementing advanced search techniques. Methods like **Query Expansion & Reformulation** will enhance the system's ability to understand and process user queries more effectively. This will lead to more accurate and relevant results by broadening or refining search terms. These improvements will reduce retrieval time, minimize hallucinations, and ensure the system delivers precise lesson plans, even for complex or ambiguous queries. Here are some examples of these techniques:

Improving Indexing and Search Efficiency

Before Optimization: When a user searches for *"Music Blocks notes,"* the system scans the entire dataset sequentially, leading to slow responses.

After Optimization: Implementing vector-based indexing (e.g., FAISS or HNSW) enables fast retrieval by mapping lesson data into high-dimensional vectors, allowing quick lookup and reducing search time.

Query Expansion

Original Query: "Loops in Music Blocks"

Expanded Query: "Loops, iterations, repeat blocks, Music Blocks programming."

This improves recall by covering different ways users might phrase the same question.

Query Reformulation

Original Query: "How do I make a song?"

Reformulated Query: "How to create a melody in Music Blocks?"

This tailors the search to the Music Blocks context, providing more relevant responses.

- **Introduce Context Filtering and a Better Chunking Strategy.**

  One more interesting thing we can do to optimize the system is to **introduce Context Filtering and a Better Chunking Strategy** to improve the accuracy and relevance of retrieved information in the **RAG system**. **Context Filtering** will ensure that only the most relevant data is used during retrieval, reducing noise and improving the quality of lesson plans. **Better Chunking** will break large documents into meaningful sections, preserving context across chunks and improving retrieval precision. These techniques will minimize hallucinations, enhance response accuracy, and ensure that the system provides clearer and more contextually relevant lesson plans. Here are some chunking strategies that we'll experiment with:

  1. **Fixed-Length Chunking:** Dividing text into fixed-size chunks (e.g., 512 tokens).
  2. **Semantic Chunking:** Split text based on semantic boundaries like paragraphs, sentences, or sections.
  3. **Sliding Window Chunking:** Creating overlapping chunks to maintain context continuity.
  4. **Recursive Chunking:** Splitting large documents hierarchically—first by section, then by paragraph, and finally by sentence.
  5. **Adaptive Chunking:** Dynamically adjusting chunk sizes based on content

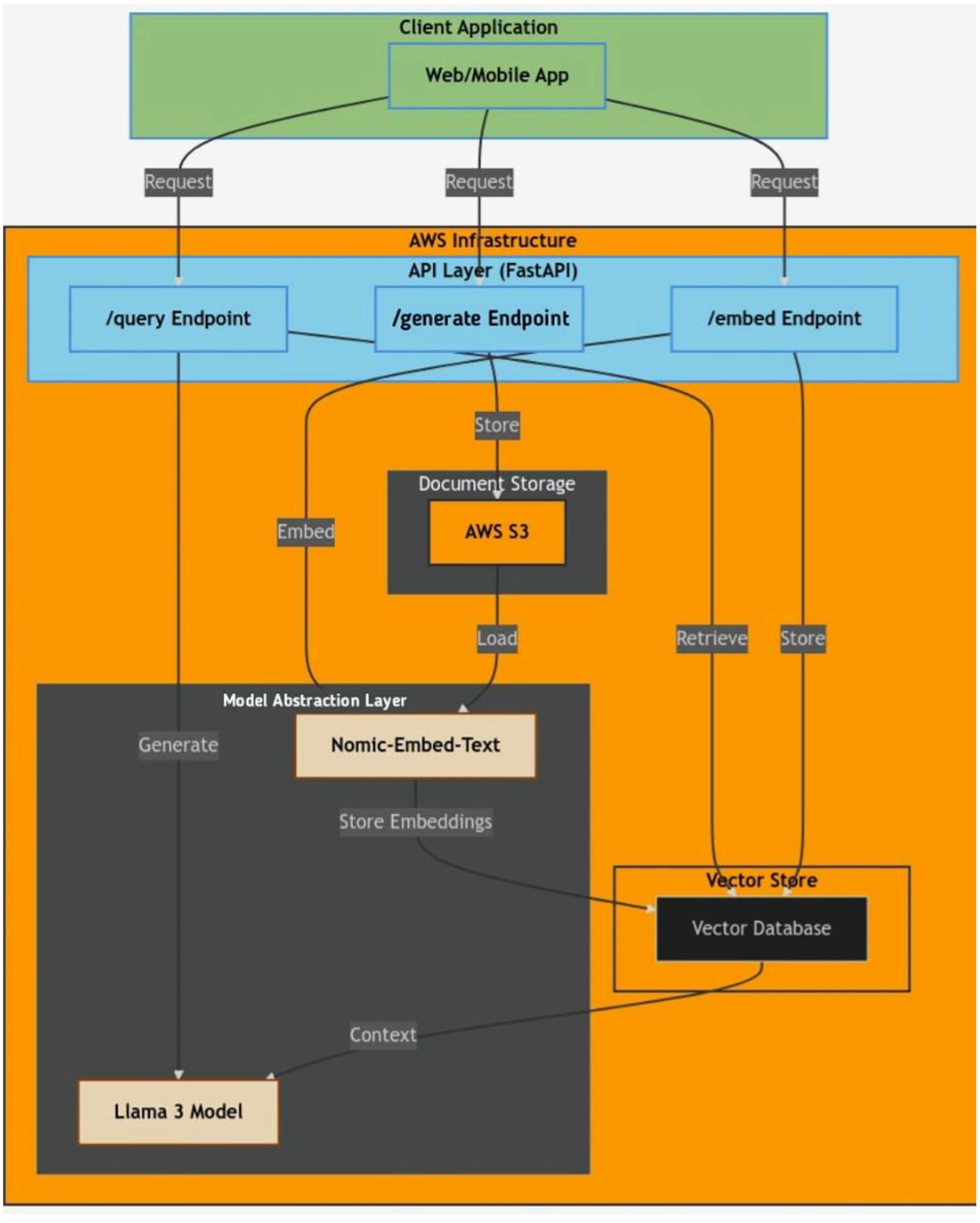
(e.g., larger for narrative, smaller for code).

6. **Query-Specific Chunking:** Adapting chunks based on the user query to retrieve the most relevant data.
7. **Metadata-Aware Chunking:** Adding metadata (e.g., page number, section header) to each chunk for better context.

- **Test and refine hallucination-reduction techniques based on performance.**

  Finally, After the implementation of the above steps, I'll be testing which technique works best and we'll move forward with the best one. By analyzing the model's responses and adjusting these techniques, we can systematically reduce hallucinations. Regular evaluations will ensure the model produces more reliable and contextually accurate lesson plans.

AI Code generation Flowchart


Client Application
Web/Mobile App
Request
Request
Request
AWS Infrastructure
API Layer (FastAPI)
/query Endpoint
/generate Endpoint
/embed Endpoint
Store
Document Storage
AWS S3
Embed
Load
Retrieve
Store
Model Abstraction Layer
Generate
Nomic-Embed-Text
Store Embeddings
Vector Store
Vector Database
Context
Llama 3 Model

Possible Edge Cases

Here are some potential edge cases to consider when implementing the code generation and abstraction layer for the lesson plan generator:

1. Ambiguous User Inputs

- **Case:** The user provides vague or unclear prompts (e.g., "Explain note").
- **Challenge:** The system may generate inaccurate or overly generic lesson plans without proper context.
- **Solution:** Implement query expansion to rephrase and clarify ambiguous inputs before processing.

2. Context Overload

- **Case:** Users provide a large number of prompts, exceeding the model's context window.
- **Challenge:** The model may forget earlier inputs or provide inconsistent responses.
- **Solution:** Implement a sliding window for context management to maintain the most relevant information.

3. Irrelevant Context Inclusion

- **Case:** Including all previous user inputs may lead to noisy, irrelevant context.
- **Challenge:** The model may hallucinate by combining unrelated information.
- **Solution:** Use context filtering to prioritize and retain only essential inputs for generating responses.

4. Incomplete or Missing Music Blocks Data

- **Case:** If Music Blocks project data is incomplete or missing from the database.
- **Challenge:** The model may fabricate information to fill the gaps.
- **Solution:** Implement data validation checks and fallback mechanisms to alert users of missing content.

5. Complex Multi-Step Queries

- **Case:** Users request lesson plans that involve multiple interconnected topics (e.g., "Explain chords and pitch block with examples").
- **Challenge:** The model may fail to maintain coherence across multiple concepts.
- **Solution:** Introduce advanced chunking strategies to break down multi-step queries and process them sequentially.

6. Edge Cases in Query Reformulation

- **Case:** Over-simplifying or over-complicating a reformulated query.
- **Challenge:** This can lead to losing important context or generating irrelevant results.
- **Solution:** Use feedback loops to assess the impact of reformulation and refine the query logic accordingly.

7. Handling Special Cases in Music Blocks

- **Case:** Certain Music Blocks concepts may not be well-documented or are highly niche.
- **Challenge:** The model may generate inaccurate or fabricated information.
- **Solution:** Identify and tag special cases to trigger additional validation or

fallback responses.

8. Handling Partially Completed Prompts

- **Case:** Users provide incomplete input (e.g., "Create a lesson plan on...").
- **Challenge:** The system may produce speculative or unrelated outputs.
- **Solution:** Use prompt clarification routines to request additional information before processing.

9. Edge Cases in FastAPI Deployment

- **Case:** High user concurrency or long-running queries.
- **Challenge:** Performance may degrade, leading to timeouts or partial responses.
- **Solution:** Implement asynchronous processing and caching for faster response times under load.

These are the few edge cases I have in my mind. I will discover and discuss more edge cases with mentors and will consider the implementation for them. Also, some edge cases may arise when we implement the functionalities mentioned above. Those edge cases can be considered at the same time of implementation.

Note: While making this proposal, I have taken the reference from the most impactful RAG papers. Along with this, I've used the proposal template provided by the Sugar Labs.

Implementation Plan

GSoC is around about 12 weeks in duration. I will be spending **80% of the time on implementing the functionalities** in this project,**10% of the time on fixing the bugs** left out in the current version of the project, and the **remaining 10% of the time on testing the code generation in the lesson plan RAG system and preparing the Wiki and writing documentation for the project**.
The detailed timeline is linked below.

Timeline	Start Date	End Date	Task
Community Bonding	8 May	01 June	Further requirements gathering, reading docs, and getting familiar with the codebase, project set-up
Coding Period starts	01 June	8 June	List down the data requirements and gather and process the necessary data.
	8 June	10 June	Improve the database by adding more Music Blocks (MB) project data.
	11 June	16 June	Update RAG prompts to integrate MB-specific code efficiently.
	17 June	1 July	Integrate context fetching from the newly added MB project data with the LLM for code generation.
	2 July	10 July	Finish the code generation feature to provide better starter code snippets to guide project creation.
	11 July	15 July	Improve code readability with syntax highlighting, indentation, comments, and explanations.
Phase 1 Evaluation	14 July	18 July	

	16 July	18 July	Add a feature to Include references to MB concepts where relevant.
	19 July	24 July	Make code easily copyable and usable in MB with export options.
	25 July	31 July	Define the Abstraction Layer for the system.
	1 August	5 August	Implement Concrete Models and classes.
	6 August	13 August	Integrate the Abstraction Layer into the RAG System
	14 August	16 August	Add Configurations and test the RAG system with different LLMs.
	17 August	19 August	Optimize retrieval mechanisms using ANNs.
	20 August	21 August	Improve indexing and search efficiency with techniques like Query Expansion & Reformulation.
	23 August	27 August	Test and refine hallucination-reduction techniques (ie. context filtering) based on performance.
Phase 2 Evaluation	24 August	1 September	
	28 August	1 September	Preparing Documentation, Wiki, and FAQs, and a Webcast on the Final Product.

Future Work

In the future, I am going to work on

1. Deploying real-time collaboration of Music Blocks.
2. Maintaining the Lesson plan project and other AI projects.
3. handling some other edge cases that may arise when actual users use this and provide feedback.

I can assure you that if I get selected to work at Sugar Labs this summer, I definitely will do my best to make this project successful and would love to continue working with Sugar Lab's other projects even after the summer.

Also for some reason, if I am not selected this year even then I'll contribute to this and other projects as much as possible and retry again next year.

Looking forward to working with us.

Thanks, And Regards
Ajeet Pratap Singh