Music Blocks 4 Program Engine

GSoC 2025 | Sugar Labs

Basic Information

Project Details

Project Name	Music Blocks 4 Program Engine
Skills	TypeScript 5, Vitest, Vite, Jest, Object-Oriented Programming, Abstract Syntax Trees
Mentors	Anindya Kundu, Walter Bender, Devin Ulibarri
Project Size	350 Hours

Personal Details

Name	Ravjot Singh
Date of Birth	07-04-2004
Country	India (Nationality & Current Residence)
Email	ravu2004@gmail.com
Resume	Resume
GitHub	GitHub
CodeForces	CodeForces
Leetcode	LeetCode
Linkedin	Linkedin
Matrix	ravjot07
University	Atal Bihari Vajpayee Indian Institute of Information Technology and Management
Course	Bachelor of Technology (BTech) in Computer Science & Engineering
Contact Number	(+91) 8727954660
Time Zone	UTC+05:30 (Asia/Kolkata)
Preferred Language	English

Introduction

I'm Ravjot Singh—a passionate developer from Chandigarh, Punjab—with a deep commitment to the open source community. Over the past years, I've actively contributed to projects that make a real impact on education and technology. My journey with sugar labs began with contributions, where I enhanced projects such as **Music Blocks and Connect the Dots, and many more sugar activities** optimizing codebases and introducing new features to enrich children's interactive learning experiences. Working with Sugar Labs has not only deepened my technical expertise but also reinforced my belief in the transformative power of collaborative development.

In addition to my Sugar Labs contributions, I have also been an active member of the **Confidential Computing Consortium at Verasion,** where I implemented critical improvements like **command functionality enhancements, continuous integration pipelines, and security validations**. These experiences have allowed me to work on both the front-end and back-end challenges of modern open source projects, ensuring robust performance and reliability.

Furthermore, my role as an **LFX Mentorship Project Contributor** at the **Cloud Native Computing Foundation on the Kmesh** project has provided me with invaluable insights into designing and deploying **end-to-end testing strategies, streamlining CI workflows, and mentoring new contributors**. This mentorship experience not only honed my technical skills but also reinforced my passion for making a difference through open source innovation.

My motivation to contribute to open source stems from the desire to build accessible, cutting-edge educational tools that empower learners—especially children—to explore, experiment, and develop critical thinking skills. Leveraging my experience with Sugar Labs' vibrant educational ecosystem and my commitment to continuous improvement, I am eager to refine and advance the Music Blocks 4 Program Engine. By enhancing its AST-based execution engine, handling concurrent time-based instructions, and optimizing its performance, I aim to ensure that Music Blocks remains a groundbreaking platform for creative music composition and interactive learning.

Contributions to Sugar Labs

As an active contributor to the Sugar Labs community, I have worked across multiple repositories including **Music Blocks**, **Connect the Dots**, **Mateton**, and various game-based Sugar activities. My contributions span UI improvements, bug fixes, code refactoring, performance enhancements, and documentation updates. I've focused on making the user experience more intuitive and the codebase more maintainable, while also contributing new features and fixing critical gameplay and rendering bugs. These efforts reflect my dedication to creating impactful, child-friendly educational tools.

PR	Repository	Contribution Summary
#3	`mateton-activity`	**Remove simplejson dependency and use standard library json instead**

#2	`fourier-series-visualisation`	**feat: display real-time frequency and circle count in UI**
#1	`fourier-series-visualisation`	**Fix: Prevent "width greater than radius" error in Pygame circle drawing**
#21	`connect-the-dots-activity`	**Updated README with clearer installation and contribution instructions.**
#26	`connect-the-dots-activity`	**Feat: Add Undo and Redo functionality**
#24	`connect-the-dots-activity`	**Fixes #23: Updated- lib/sugar-web**
#22	`connect-the-dots-activity`	**Refactored redundant JavaScript code to enhance maintainability.**
#19	`connect-the-dots-activity`	**Added clear button and refractored code**
#4150	`musicblocks`	**Add: Test for utils.js**
#4117	`musicblocks`	**Feat: Added live visual feedback for audio input in Sampler Widget**
#42	`make-them-fall-activity`	**Fix: Activity Start error**
#18	`hittheballs-activity`	**Resolved ball movement issues after game reset.**
#562	`www`	**Fixed typos**

Besides my work with Sugar Labs, I have also contributed to other significant open-source projects, including **Veraison** under the **Confidential Computing Consortium** and **Kmesh** under the **Cloud Native Computing Foundation (CNCF)**.

My Contributuions to Project Version:

PR	Repository	Contribution Summary
#14	`cocli`	**Modified command functionality to support unsigned CoRIMs, improving system compatibility.**

#10	`gen-corim`	**Developed and integrated a CI pipeline for running unit tests on each commit.**
#20	`cocli`	**Authored detailed documentation for CoMID templates and extension points.**
#22	`cocli`	**Redesigned display options for enhanced modularity and maintainability.**
#143	`corim`	**Implemented MAC/IP address validation in Measurement.Valid to enhance reliability and security.**

My Contributuions to Project Kmesh:

PR	Title	Description
#1277	**Add E2E test for Locality Load Balancing**	**Developed an end-to-end test validating locality-based traffic distribution, ensuring correct behavior under load balancing rules.**
#1243	**feat(authz): Add E2E test for kmeshctl authz functionality**	**Implemented tests for authorization CLI commands, improving coverage and reliability of `kmeshctl`.**
#1238	**feat: Add readiness probe for Kmesh daemon**	**Added a readiness probe to ensure safe and stable daemon startup before traffic is accepted.**
#1231	**Fix CI: Regenerate docs for authz subcommands**	**Fixed CI issues by updating auto-generated documentation for authorization commands.**
#1226	**Add Unit Tests for logs Package**	**Introduced unit tests to increase coverage and validate logging functionality.**

PROJECT IDEA

Synopsis:

This proposal outlines a plan to **refine and complete the execution engine for the Music Blocks 4 Program Engine under Sugar Labs as part of Google Summer of Code (GSoC) 2025.** The project is aimed at enhancing an in-memory AST-based interpreter to support both imperative and declarative syntax, along with robust time-based instruction execution and concurrent thread management. By leveraging modern TypeScript features, optimized scheduling, and a plugin-based architecture, this work will significantly improve the Music Blocks program's responsiveness, scalability, and maintainability. The final outcome will be a high-performance, production-ready engine that integrates seamlessly with Sugar Labs' educational ecosystem and fosters innovative, interactive music programming experiences.

Summary of the Project:

Music Blocks is an educational platform designed to allow children to create music through visual programming. While previous iterations have provided a functional baseline, critical improvements are needed to achieve a production-ready engine capable of managing complex, time-based musical instructions and concurrent execution. This project will focus on the following key components:

- **AST-based Execution Engine:** Implement an in-memory interpreter that parses and executes an Abstract Syntax Tree (AST) representing Music Blocks programs. The AST will support both imperative commands and declarative instructions for handling musical events.

- **Parser and Interpreter:** Develop a modular parser that traverses the AST and an interpreter that executes each node accordingly. These components will work together to process nested control structures, function calls, loops, and real-time operations.

- **Concurrent Thread Execution:** Introduce a robust scheduler to manage concurrent threads (or processes) representing multiple musical sequences. The scheduler will implement cooperative multitasking to handle time-based instructions without stalling the user interface.

- **Time-based Instruction Handling:** Incorporate mechanisms to accurately schedule and execute time-sensitive instructions, leveraging asynchronous patterns (using async/await, Promises, and event loop coordination) to ensure precise timing for note playback and other musical actions.

- **Tooling and Ecosystem Integration:** Utilize TypeScript 5, Vite, and Vitest to modernize the project's codebase, ensuring high performance, clear architecture, and smooth developer experience. The revised engine will also align with Sugar Labs' UI/UX guidelines for a cohesive experience in the Sugar desktop environment.

Throughout the project, clear abstractions will be established to support future extensions, allowing plugin developers to add new syntax elements or musical operations with minimal changes to the core engine. The approach not only aims to enhance performance and maintainability but also to provide a solid template for building similar AST-driven, time-synchronized execution engines.

What will be Achieved:

Upon completion of this project, the following outcomes will be delivered:

- **Refined AST-Based Interpreter:** A fully functioning in-memory interpreter that effectively handles both imperative and declarative syntax, specifically optimized for the intricacies of musical programming.

- **Robust Scheduling and Concurrency Model:** An integrated scheduler supporting concurrent thread execution and precise time-based scheduling. This ensures multiple musical sequences and time-based events execute seamlessly without blocking the main event loop.

- **Modular, Object-Oriented Architecture:** Clean, well-documented, and extensible codebase that establishes best practices for AST node representations, parser/interpreter separations, and state management. This architecture will facilitate future enhancements and module reuse in other Sugar Labs projects.

- **Comprehensive Testing and Documentation:** A full suite of unit and integration tests using Vitest, along with detailed developer and user documentation. This ensures reliability for future contributors and provides guidance for maintaining and extending the engine.

- **Performance Optimization:** Practical optimizations to minimize overhead in AST traversal, manage state and scheduling effectively, and ensure the interpreter performs well in resource-constrained environments (such as the Sugar on XO laptops).

- **OSS Engagement and Long-term Sustainability:** A clear contribution pathway with documented best practices and design patterns, allowing the Sugar Labs community to maintain and build upon this work. This includes guidelines for collaboration, code review processes, and integration with existing Sugar Labs repositories.

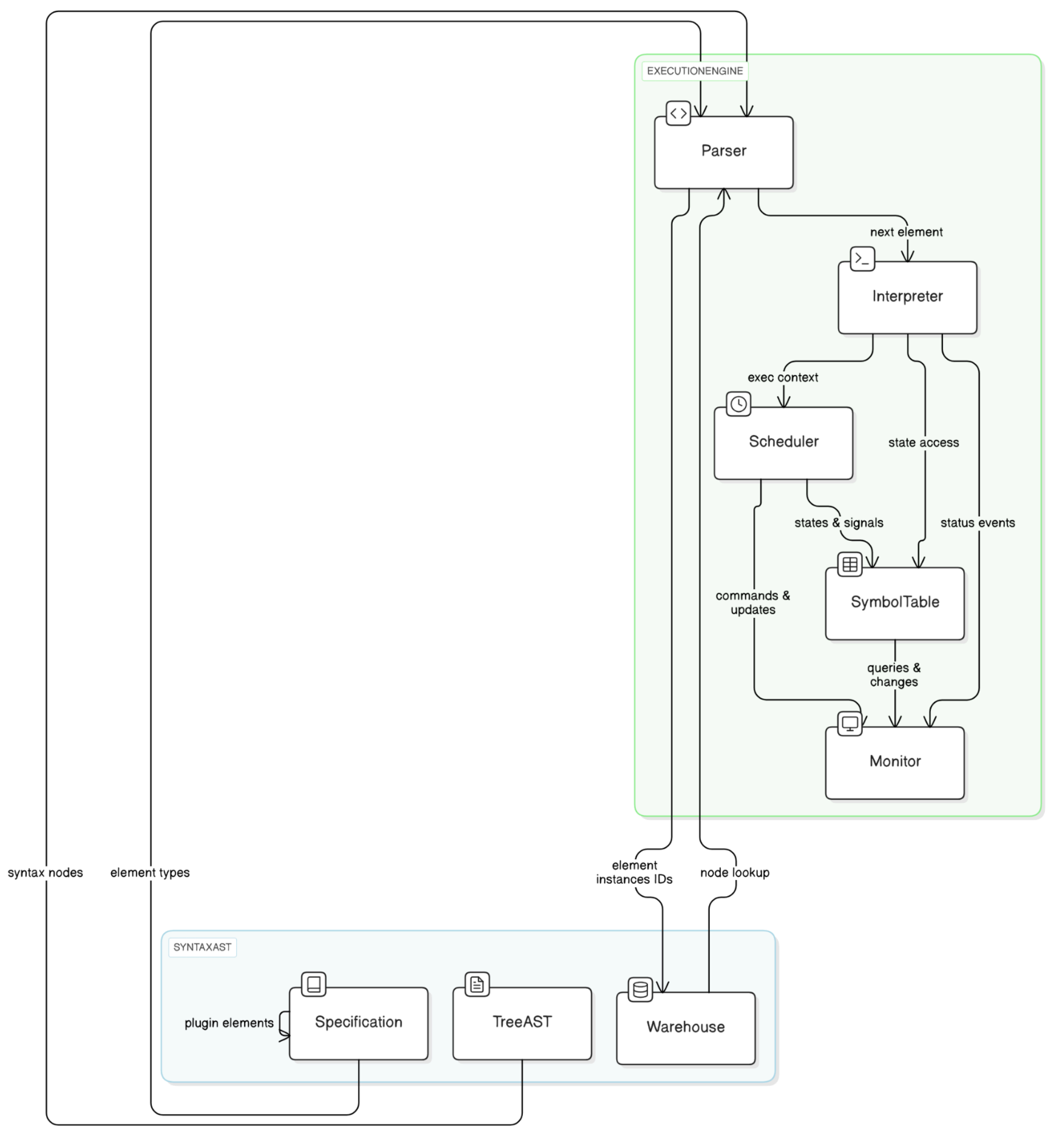
By the end of GSoC, the project will have delivered a polished, production-ready Music Blocks execution engine that enhances the creative and educational potential of the platform. This engine will not only satisfy immediate project requirements but also provide a strong, extensible foundation for future developments in music-based educational programming.

Deliverables:

Here is a proposed **modular execution engine architecture** for Music Blocks 4, along with implementation strategies and references to inform each component:

- **AST Structure (Tree and Nodes):** Use an in-memory AST (`Tree`) to represent the program's block structure github.com . Each block or syntactic element is a node in this tree, with references to child nodes (for blocks that contain statements or expressions). The AST can be dynamically updated if the user edits the program during execution (Sugar Labs might want live coding support), but typically it's constructed at program start from the saved project. The architecture already defines an `Element API` for syntax elements and a `Specification` registry of element types github.com. Implementation will involve defining TypeScript classes or interfaces for all element types (including core ones like **Process** and **Routine** as special block types github.com. Plugins will extend these classes. Each node class will implement methods needed for execution (e.g., an `execute` method, and possibly utility methods for the parser like `getChildren()`).

Detailed Flowchart for Syntax & AST and Execution Engine



Architecture of the Music Blocks Program Engine. The **Syntax & AST** subgraph (green) represents the static program structure: a Tree of connected syntax elements, definitions (Specification) of all element types (including plugins), and a Warehouse tracking instances. The **Execution Engine** subgraph (blue) represents runtime components: the Parser traverses the AST (with help from Specification/Warehouse to map node IDs to instances), the Interpreter executes each element's behavior (interacting with the global state in the Symbol Table), and the Scheduler manages multiple concurrent execution threads (processes) and timing. The Monitor observes state and execution events to provide feedback (e.g., for UI highlighting or debugging). This architecture cleanly separates concerns: modifications to the AST or adding new block types (via plugins) do not require changes in the core execution logic, and concurrency/timing is handled by the Scheduler component.

- **Parser (Execution Orchestrator):** The Parser will be responsible for stepping through the AST in the correct order. It maintains a **program counter** and **call stack** github.com. In practice, this means:
    - If a Routine (function) is called, the parser pushes a new frame (saving the return address and local variable context) and then starts executing the routine's body.
    - If a loop is encountered, the parser might handle the looping construct (e.g., by looping internally or by re-inserting the loop body back into the execution sequence multiple times).
    - The parser likely works closely with the **Interpreter**: it might provide the interpreter with the "next" element to execute. Alternatively, the parser itself could be integrated into the interpreter's control flow (for example, each node's execute method could internally call the parser for substructure). We will clarify this by implementing a clear contract: perhaps the parser has a method like `step()` that returns the next `SyntaxElement` to execute (and advances the program counter), and the interpreter calls `step()` repeatedly.
    - This separation is reminiscent of how some virtual machines separate the code-fetch stage from execution. It could help in managing things like breakpoints or single-stepping in the future (the Monitor could use the parser's program counter to know where we are).
- **Interpreter (Executor):** The Interpreter takes elements from the parser and performs the actions. For a **Data/Expression** element, it will compute a value (using the current state). For an **Instruction** (statement/block), it will carry out the command, which might modify the state or produce output (sound, etc.). For example:
    - A "Play Note" instruction node, when executed, would call the appropriate sound generation function (perhaps interfacing with Web Audio or another system) to play the specified note. It might also interface with the Scheduler to schedule the note's duration (e.g., schedule stopping the note after a certain time).
    - A control structure like "Repeat 4 times" (loop) when executed might not directly loop itself, but instead signal to the parser to repeat a section. Alternatively, the interpreter could implement the loop by moving the program counter backwards. This design decision will be made during implementation: a simple approach is to have the interpreter detect loop nodes and handle looping internally (maintaining a loop counter and using a while loop in the execute method of

that node).

  - The interpreter also interacts with the **Symbol Table** (State) for variable assignments, lookups, and maintaining any temporary state needed by elements (for instance, a loop might use the symbol table to store the current iteration count if it's a for-loop with a named index).

  - Errors or edge cases (like division by zero in an expression, or an unknown block type) should be handled gracefully, possibly by throwing exceptions or emitting error events that the Monitor can catch to inform the user.

- **Scheduler (Concurrency Manager):** The Scheduler component enables multiple processes to run. In Music Blocks terms, a **Process** is an independent thread of execution (like a parallel timeline), and multiple processes can execute concurrently github.com. The scheduler will manage a list of active processes and ensure they advance over time:

  - If using a cooperative tick loop, the Scheduler will periodically call the parser/interpreter to advance each process by a step or a few steps. The Scratch forum description provides a guide: execute a chunk of blocks in one process, then switch to the nextscratcharchive.asun.co . A chunk might be defined as "run until a yield is needed or the end of a loop iteration" scratcharchive.asun.co The scheduler needs to handle **wait timing**: when a process hits a time-based wait (or a "pause until" event), the scheduler should note the target time or condition, and **suspend** that process until the time/condition is reached. For time waits, this could be done by using `setTimeout` to mark the process ready at a future time, or by calculating the wake-up time and checking it each tick. The architecture suggests the scheduler provides an *execution context* and manages orchestration github.com, so it likely will own the main loop that drives execution.

  - If leveraging `async/await`, the scheduler might be implicit – e.g., launching each process as an async function (which internally awaits on delays) and leaving it to the JS runtime to interleave. However, an explicit scheduler loop can give more control (especially for synchronizing processes on musical beats or for visualizing execution). A hybrid approach might also work: use async/await for simplicity in coding each process's logic, but still have a scheduler that initiates and monitors these async processes (and perhaps uses `Promise.race` or similar to react when any process completes a step or needs attention).

  - **Thread safety** is less of a concern in JS (since it's single-threaded), but logical conflicts can occur (e.g., two processes writing the same variable). The interpreter should document that and possibly avoid global shared state where not necessary, or use locks if truly needed (locks likely not needed if careful – simpler is to let them interleave arbitrarily as Scratch does, leaving it to the program logic to avoid races).

- **State Manager (Symbol Table):** This structure maintains variables, perhaps in multiple scopes. It can be as simple as a nested dictionary (e.g., a stack of scope objects for each call/process). Provide methods: `declare(name, value, scope)` for creating a new variable (or setting initial state), `get(name)` and `set(name, value)` to retrieve/update. If music blocks allows dynamic creation of variables or property lists (like lists of notes), the symbol table should accommodate that (maybe storing complex objects). Additionally, the symbol table might keep **special states**: e.g., current tempo, current octave, etc., that instructions might query or set. The architecture also mentions "states" and "queries" between Symbol Table and other components github.com – possibly meaning that elements can query the symbol table for current state (like the state of a toggle, or sensor input if any).

- **Monitor and Feedback:** While not the core of execution, it's worth noting that a Monitor is planned to track execution state and statistics github.com. This could be implemented by emitting events from the interpreter (e.g., "NoteOn", "NoteOff", "LoopStart", "LoopEnd", "ProcessFinished" events). The Monitor can listen and update the UI (like highlighting the current block, or showing a visual indicator of threads). Designing the interpreter with hooks (using an event emitter or callbacks) will make it easier to integrate debugging and visualization tools, which are important in educational software. For instance, each time a block is executed, the interpreter could call `monitor.blockExecuted(blockId)` to allow the front-end to highlight that block.

- **Reference Implementations:** This architecture aligns with patterns seen in Scratch VM and others:

    - Scratch's **Threading Model** – each script corresponds to a thread managed by a scheduler groups.google.com.

    - **Concurrency in Co** (a toy language with coroutines): that interpreter captured continuations at `yield` points and scheduled them, effectively splitting execution across multiple tasks abhinavsarkar.net abhinavsarkar.net. Music Blocks can follow a similar approach using either explicit continuation objects or by structuring code with async/await to naturally yield.

    - **ChucK and Sonic Pi** – emphasize scheduling and accurate timing, which justify having a dedicated scheduler and possibly an internal clock to coordinate events.

    - Any time-based AST execution (such as game scripting engines) often have a main loop that updates each script each frame – a similar pattern will serve Music Blocks well for updating musical timelines.

- **Performance Considerations:** With the above architecture, we should also plan for optimizations:

    - If many blocks are no-ops (e.g., data blocks just returning values), the overhead of visiting them could add up. Caching constant values or folding constant expressions when loading the AST can reduce runtime work.

    - The use of **Web Workers** could be explored for heavy tasks, but likely the engine will run fast enough in the main thread given typical program sizes. We can mention that for extreme cases (like generating audio waveforms or handling large data sets), a worker could offload the work to prevent UI jank medium.com.

- Use profiling tools in Chrome/Node to find bottlenecks once the engine is running, and address them (e.g., if the dynamic dispatch on node types is slow, consider switching to a lookup table of functions by opcode).

- Garbage generation should be minimized in the inner loop of execution. For example, the scheduler can maintain a pool of "thread" objects instead of recreating them for each cycle.

The outcome of this architecture will be a **robust execution engine** that can parse Music Blocks programs and execute them with correct semantics, including support for **concurrent processes and precise musical timing**. It will be extensible (new block types via plugins), debuggable (with monitor hooks), and maintainable due to clear separation of components.

Proposed Timeline:

The project is planned for the GSoC 2025 timeline (350-hour project). Below is a week-by-week breakdown, including the community bonding period, planning, development milestones for each activity, testing phases, progressive development milestones for key components, integration and testing phases, and dedicated time for code polishing and comprehensive documentation. The timeline is tentative and may be adjusted in consultation with mentors, but it demonstrates a clear iterative approach: a basic version will be implemented early, then refined through multiple iterations to add robust features, comprehensive testing, and performance optimizations before the final evaluation.

Timeline	Activity	Key Tasks / Deliverables
Community Bonding *(May 8 – May 28)*	**Community Engagement & Setup**	**Community Engagement & Environment Setup** ☑ ~~- Set up development environment: configure Sugar desktop, sugar-toolkit, and necessary TypeScript libraries~~ ☑ ~~- Familiarize with the existing Music Blocks-v4 codebase~~ - Discuss initial design ideas, review current documentation, and finalize execution engine specifications with mentor input - Draft preliminary UI/UX and system diagrams (using Mermaid.js) - Plan testing strategies and collect necessary assets
Week 1 *(May 29 – June 4)*	**AST Design and Specification**	- Define data structures for AST nodes using a class hierarchy and/or discriminated unions in TypeScript - Establish the Specification registry for syntax elements (including plugins) - Create UML/Mermaid diagrams for the AST architecture

		- Validate design through mentor review and discussion
Week 2 *(June 5 – June 11)*	**Basic Parser Implementation**	- Develop the initial parser to convert Music Blocks programs into an in-memory AST - Integrate AST node definitions and basic error handling - Write unit tests to verify accurate AST generation - Document parsing decisions and initial challenges
Week 3 *(June 12 – June 18)*	**Interpreter Core Implementation**	- Implement the execution engine that traverses the AST and executes node-specific behaviors - Establish support for imperative constructs (loops, conditionals, and simple function calls) - Write unit tests for early-stage execution scenarios
Week 4 *(June 19 – June 25)*	**State Management & Scheduler Introduction**	- Develop a state manager (symbol table) to handle program variables and global musical parameters (e.g., tempo) - Introduce an initial scheduler using async/await to manage time-based instructions - Create unit tests to ensure reliable state tracking and basic scheduling
Week 5 *(June 26 – July 2)*	**Time-Based Instruction Handling**	- Enhance the interpreter to support time-scheduled (delay/wait) instructions - Integrate these with the scheduler to yield control and resume execution precisely - Validate timing accuracy with both manual and automated tests
Week 6 *(July 3 – July 9)*	**Concurrency and Multi-Thread Execution**	- Extend the scheduler to manage multiple concurrent execution threads (simulated processes) - Implement cooperative multitasking to allow simultaneous musical sequences - Introduce synchronization primitives (e.g., “broadcast and wait”) and test concurrent scenarios
Week 7 *(July 10 – July 16)*	**Optimizations and Advanced Features (Part 1)**	- Profile the interpreter and scheduler to identify performance bottlenecks - Optimize AST traversal (e.g., caching or iterative loops in place of recursion) - Refine state management for improved runtime efficiency - Update unit tests for performance improvements
Week 8 *(July 17 – July 23)*	**Advanced Features and Extensibility (Part 2)**	- Enhance support for declarative constructs alongside imperative execution - Refine object-oriented design and plugin system to easily extend functionality with new block types - Incorporate mentor feedback for additional features and improvements

Week 9 *(July 24 – July 30)*	**Comprehensive Integration Testing**	- Integrate the parser, interpreter, and scheduler into a cohesive execution engine - Run full-scale tests using complete Music Blocks programs - Validate concurrency and timing through scenario-based integration tests - Address integration bugs and refine error handling
Week 10 *(July 31 – Aug 6)*	**Documentation and Code Refinement**	- Develop complete technical documentation covering the design, API, and user guide for the execution engine - Draft developer guides and detailed diagrams (using Mermaid.js) - Enhance inline code comments and prepare design rationales for future maintainers
Week 11 *(Aug 7 – Aug 13)*	**Final Polishing, Mentor Review, and Buffer**	- Incorporate final mentor feedback and perform extensive code refactoring - Conduct thorough end-to-end testing (unit and integration tests) - Optimize performance further and address any remaining issues - Prepare demonstration materials and final report
Week 12 *(Aug 14 – Aug 21)*	**Submissions and Wrap-Up**	- Finalize code base, documentation, and demo videos - Merge final code into Sugar Labs repositories - Complete final testing and quality assurance - Submit final project deliverables and evaluations

Availability

I am committed to contributing approximately **40 to 50 hours per week** to the project.. I will ensure consistent participation and focused work throughout the program to achieve all project goals and deadlines effectively.The GSoC timeline aligns well with my university's **summer break**, giving me ample uninterrupted time to focus on the project. Even after the break ends, I will have **no exams or in-person classes**, and all academic commitments will be **asynchronous**, allowing me to continue contributing actively.

My typical working hours are from **10:00 IST (4:30 UTC) to 1:00 IST (19:30 UTC)**, during which I will be fully available and responsive. In case of any unforeseen circumstances requiring time off, I will promptly inform my mentors in advance.

Post GSoC Plans

After GSoC, I plan to remain an active contributor to Sugar Labs and continue refining the Music Blocks 4 Program Engine. Building on the progress made during the project, I intend to extend the engine's capabilities by exploring additional optimizations, integrating new block types, and enhancing concurrency and scheduling techniques. In particular, I aim to mentor new contributors who take on similar challenges, assist with code reviews, and guide future design enhancements as feedback is received from the community. I also plan to share insights and experiences through blog posts, community meetings, and technical talks—helping to disseminate best practices in building educational programming tools that combine music and interactive learning. This continued involvement will not only maintain the engine's evolution but also foster a vibrant, collaborative development culture within Sugar Labs.

Conclusion

The Music Blocks 4 Program Engine project is much more than an exercise in refining a codebase—it is an opportunity to empower creative learning by blending the art of music with the logic of programming. With this project, we are set to build a robust, in-memory AST interpreter that supports both time-based instructions and concurrent execution, ultimately enhancing the interactive experiences that define Sugar Labs' educational ecosystem.

My background in TypeScript development, object-oriented design, and contributions to Sugar Labs has provided me with a strong foundation to tackle this complex challenge. By applying modern software engineering practices and embracing a collaborative, iterative development approach, this project promises to deliver a production-ready engine that is not only high-performing and extensible but also accessible to young learners and educators alike.

I am deeply motivated to bring this vision to life. Through structured development, thoughtful integration of advanced scheduling techniques, and comprehensive testing, the Music Blocks 4 Program Engine will become a cornerstone for innovative music programming. I look forward to collaborating with mentors and the Sugar Labs community, ensuring that this project remains a valuable asset that inspires creativity and interactive learning for children around the world.