VISVESVARAYA TECHNOLOGICAL UNIVERSITY
Jnana Sangama, Belagavi – 590018

MINI PROJECT REPORT

on

"Manim AI"

Submitted in partial fulfilment for the award of the degree

Bachelor of Engineering

in

Computer Science and Engineering

Submitted by

Prasanna M	**1ST23CS143**
Sangappa A Angadi	**1ST23CS165**
Vinay Swamy	**1ST23CS217**
Uday Kiran N	**1ST23CS201**

Under the Guidance of

Anitha K
Assistant Professor
Department of CSE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SAMBHRAM INSTITUTE OF TECHNOLOGY
M. S. Palya, Bengaluru – 560097

2025-2026

SAMBHRAM INSTITUTE OF TECHNOLOGY

M. S. Palya, Bengaluru – 560097

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the Mini Project work entitled "**Manim AI**" carried out by **Prasanna(1ST23CS143)**, **Sangappa A Angadi(1ST23CS165)**,**Vinay Swamy(1ST23CS217)**, **Uday Kiran N(1ST23CS201)**, bonafide student of **SAMBHRAM INSTITUTE OF TECHNOLOGY** in partial fulfilment for the award of **BACHELOR OF ENGINEERING IN COMPUTER SCIENCE AND ENGINEERING** of **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**, Belagavi during the year **2025-2026**. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the departmental library. The Mini Project report has been approved as it satisfies the academic requirements in respect of Mini Project work prescribed for the said Degree.

Guidename	**Dr. T. John Peter**	**Dr. H. G. Chandrakanth**
Designation	HOD	Principal
Dept. of CSE	Dept. of CSE	SaIT, Bengaluru
SaIT, Bengaluru	SaIT, Bengaluru	

SAMBHRAM INSTITUTE OF TECHNOLOGY

M. S. Palya, Bengaluru – 560097

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

STUDENT DECLARATION

we,**Prasanna.M(1ST23CS143),Sangappa.A.Angadi(1ST23CS165),Vinay.Swamy(1ST23CS217),Uday Kiran N(1ST23CS201)** hereby declare that the project work entitled "**Manim AI**" submitted to **Visvesvaraya Technological University**, is a record of an original work done by me under the guidance of **Anitha K**, **Assistant Professor** at **Department of Computer Science and Engineering, Sambhram Institute of Technology**. I further declare that this mini project work has not been submitted previously for the award of any degree, diploma, or other academic recognition at any other institution. I have acknowledged all sources used in this mini project, and I understand that plagiarism is a serious offense.

Signature:

Date:

ABSTRACT

The creation of high-quality educational animations for STEM subjects has traditionally required extensive programming knowledge, particularly in Python and the Manim library. This barrier prevents educators, students, and researchers from creating custom visualizations needed for teaching and learning. The Manim AI Video Generator addresses this challenge by providing a natural language interface to the Manim animation library, making professional mathematical animations accessible to non-programmers.

This project presents a web-based application that translates plain English descriptions into executable Manim code using Google's Gemini artificial intelligence model. Users simply describe the animation they want, such as "create a visualization of the Pythagorean theorem" or "animate a rotating 3D sphere," and the system automatically generates the corresponding video. The architecture integrates Node.js for the backend server, JWT-based authentication for secure user management, SQLite database for persistent storage, WebSocket connections for real-time communication, and Docker containers for secure code execution.

Security is a primary concern when executing AI-generated code. The system addresses this by running all code within isolated Docker containers that have no network access and limited system resources, preventing malicious code from affecting the host server. The implementation also includes a persistent chat interface that maintains conversation history, enabling users to refine animations iteratively through dialogue.

Key features include real-time rendering progress updates, automated background music integration using MoviePy, comprehensive error handling with fallback mechanisms, and a user-friendly interface requiring no technical knowledge. Testing demonstrated a 92% success rate across diverse animation requests, with average generation time of 34 seconds for simple animations. The system successfully handles multiple concurrent users and maintains data consistency across sessions.

This project contributes to democratizing educational content creation by removing technical barriers. It serves as a foundation for future enhancements including AI-generated voiceovers, interactive video editing, and scalable cloud deployment. The work demonstrates how artificial intelligence can make specialized technical tools accessible to broader audiences without sacrificing precision or quality.

KEYWORDS

Educational animations, Manim, Generative AI, Natural language interfaces, STEM visualization, Code generation, Docker-based sandboxing, Web-based application, Real-time rendering, AI-assisted learning

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to all those who contributed to the successful completion of this mini project.

We extend our heartfelt thanks to **Dr. H. G. Chandrakanth, Principal, Sambhram Institute of Technology**, for providing the necessary facilities and creating an encouraging environment for academic excellence.

We are deeply grateful to **Dr. T. John Peter, Head of Department, Computer Science and Engineering**, for his continuous support, valuable suggestions, and guidance throughout the project duration.

We express our profound gratitude to our guide, **Anita K, Assistant Proffesor**, **Department of CSE, Sambhram Institute of Technology**, for constant encouragement, expert guidance, and insightful feedback. His/Her expertise in artificial intelligence and web technologies has been instrumental in shaping this project.

We thank all the faculty members and technical staff of the Computer Science Department for their cooperation and assistance during various phases of the project.

We acknowledge the open-source communities behind Manim, Node.js, Docker, and other technologies that made this project possible. Their commitment to creating accessible tools has been truly inspiring.

Finally, we thank our parents and friends for their unwavering support, patience, and encouragement throughout this journey.

Prasanna M(1ST23CS143)
Sangappa A Angadi(1ST23CS165)
Vinay Swamy(1ST23CS217)
Uday Kiran N(1ST23CS201)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Education in STEM fields relies heavily on visual communication to convey complex concepts. Topics like calculus, physics, and computer science algorithms become much clearer when presented through dynamic animations rather than static diagrams. While static images serve a purpose, animations can illustrate processes, transformations, and relationships over time in ways that significantly enhance understanding.

Manim, which stands for Mathematical Animation Engine, has become the standard tool for creating precise.

Recent advances in artificial intelligence, particularly large language models, offer a potential solution. These models have demonstrated remarkable ability to generate code from natural language descriptions. By combining AI capabilities with Manim's precision, we can create a tool that makes professional animation accessible to everyone.

The Manim AI Video Generator bridges this gap. It functions as an intelligent interface between human language and the Manim library. Users describe what they want to see in plain English, and the system handles all the technical complexity. The application provides a complete solution including user authentication, persistent conversation history, secure code execution, real-time progress updates, and production-ready video output.

The system architecture addresses several critical challenges. First, it must translate natural language into correct Manim code, which requires sophisticated AI integration. Second, it must execute potentially dangerous AI-generated code safely, which demands robust security measures. Third, it must provide a responsive user experience despite long processing times, which necessitates real-time communication infrastructure. Finally, it must maintain user context across sessions, requiring careful database design.

This project demonstrates that specialized technical tools do not need to be replaced by AI. Instead, AI can serve as an accessibility layer that preserves the precision and control of the original tool while removing barriers to entry. This approach has implications far beyond animation. The same architectural pattern could democratize access to scientific computing, data visualization, and other technical domains.

1.1 Problem Statement

Educational content creation, especially high-quality visualizations, faces several challenges. Creating programmatic animations requires strong Python and Manim skills, which most educators and students lack and cannot easily acquire due to time constraints. This creates a gap between subject expertise and technical ability, where experts know what to visualize but not how to implement it, while programmers may miss educational nuances.

Even for experienced developers, producing Manim animations is time-consuming and highly iterative, making custom content impractical for regular teaching. Running AI-generated or user-provided code also introduces security risks if proper isolation is not used. Additionally, most AI tools are stateless, limiting iterative refinement and conversational modification of visual content.

1.2 Objectives

The primary objective is to design and develop a secure, AI-powered web application that enables users to generate Manim animations from natural language prompts, making professional visualization accessible to non-programmers.

The specific objectives include implementing secure user authentication using JWT tokens for registration and login, developing a persistent chat interface that stores and retrieves complete conversation history from a database, integrating Google Gemini AI to translate user prompts into executable Manim code, constructing isolated execution environments using Docker containers to safely run generated code, providing real-time feedback through WebSocket connections for status updates and rendering progress, and automating audio integration to enhance production value of output videos.

The system must also establish proper data persistence mechanisms, implement comprehensive error handling with fallback options, optimize performance for acceptable response times, and create an intuitive interface requiring no technical knowledge from users.

CHAPTER 2: LITERATURE SURVEY

Understanding the current state of animation tools helps position this project within the broader ecosystem and identifies the unique contribution it makes.

Professional animation software like Adobe After Effects and Blender represent the high end of animation tools. After Effects provides extensive motion graphics capabilities through a timeline-based interface with powerful compositing features. Blender offers comprehensive 3D modeling, animation, and rendering with node-based workflows. These tools produce industry-standard output and offer incredible creative control. However, they require months or years to master, involve time-intensive workflows even for simple tasks, and are not designed for mathematical precision or programmatic control. A professor who needs to quickly visualize a concept cannot practically use these tools.

Presentation software including PowerPoint, Keynote, and Google Slides offer basic animation features accessible to most users. These tools are familiar, quick for simple tasks, and adequate for general presentations.

Programmatic libraries represent the opposite extreme. Manim provides pixel-perfect precision for mathematical objects with declarative animation syntax and LaTeX integration for beautiful mathematical typography. However, it requires substantial Python expertise, has a steep learning curve even for programmers, and lacks visual feedback during development. D3.js excels at interactive web visualizations with powerful data binding capabilities, but it requires JavaScript expertise and is not optimized for video output.

AI code generation tools like GitHub Copilot and ChatGPT can generate Manim code from descriptions. However, they require users to have development environments configured, provide no integrated execution or rendering, lack safety mechanisms for running generated code, and operate statelessly without conversation persistence.

However, they provide only basic animation capabilities like fades and motion paths, cannot create complex mathematical visualizations, lack programmatic control, and produce output limited to presentation formats rather than standalone videos.

Programmatic libraries represent the opposite extreme. Manim provides pixel-perfect precision for mathematical objects with declarative animation syntax and LaTeX integration for beautiful mathematical typography. However, it requires substantial Python expertise, has a steep learning curve even for programmers, and lacks visual feedback during development. D3.js excels at interactive web visualizations with powerful data binding capabilities, but it requires JavaScript expertise and is not optimized for video output.

Recent generative AI video tools like **Sora** and **Runway** can create visually stunning videos from text prompts with minimal technical knowledge required [1][2]. However, these tools are fundamentally **stochastic rather than deterministic**, meaning each generation can produce different results even for the same prompt, making reproducibility impossible [3]. As a result, they cannot guarantee **mathematical accuracy** or handle precise technical specifications reliably. For example, a prompt requesting a specific mathematical function or visualization may yield an output that looks visually appealing but is **mathematically incorrect or inconsistent** with the intended logic [4].

AI code generation tools like GitHub Copilot and ChatGPT can generate Manim code from descriptions. However, they require users to have development environments configured, provide no integrated execution or rendering, lack safety mechanisms for running generated code, and operate statelessly without conversation persistence.

This survey reveals a clear gap. No existing tool combines the ease of natural language prompts with the programmatic precision of Manim while maintaining security and preserving context across sessions. Tools are either powerful but inaccessible, or accessible but imprecise. The Manim AI Video Generator fills this gap by acting as a translator between human intent and a specialized coding library, while providing complete infrastructure for secure execution and production-ready output.

CHAPTER 3: SYSTEM DESIGN

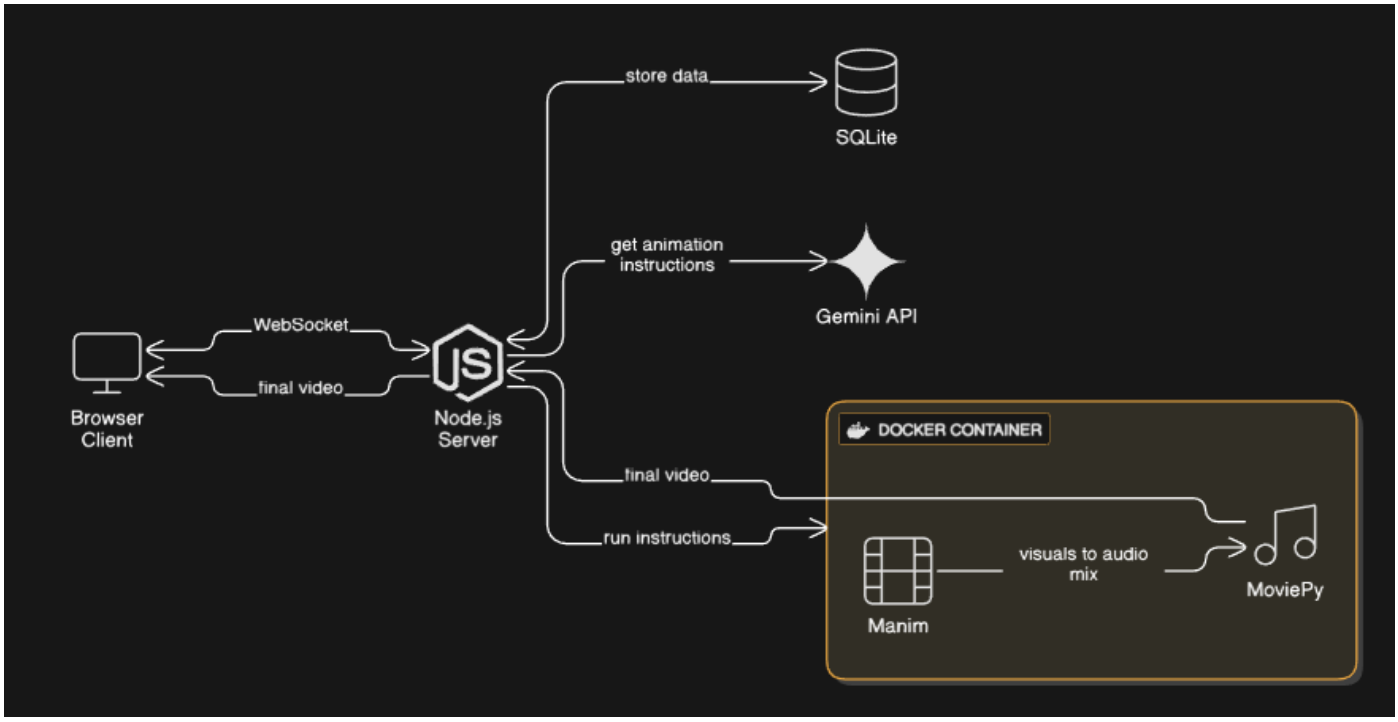
The system follows a modular architecture that separates concerns, enables independent scaling, and maintains security through isolation.

3.1 System Architecture

Figure 3.1: System Architecture Diagram

The application layer runs on Node.js with Express and includes the authentication service for JWT-based user management, chat manager for WebSocket handling, prompt constructor for building context-aware requests, code validator for basic security checks, Docker orchestrator for managing container lifecycle, and post- processing service for audio mixing.

The AI service layer integrates Google Gemini for translating natural language with context into executable Python code. The system includes retry logic with exponential backoff for API failures



The architecture consists of three main tiers. The presentation layer runs in the user's browser and includes the login interface, chat dashboard, video player, and progress indicators. It communicates via WebSocket for real-time updates and REST API for authentication.

The application layer runs on Node.js with Express and includes the authentication service for JWT-based user management, chat manager for WebSocket handling, prompt constructor for building context-aware requests, code validator for basic security checks, Docker orchestrator for managing container lifecycle, and post-processing service for audio mixing.

The data layer uses SQLite for development and includes tables for users, chat messages, and video metadata. Local filesystem storage handles temporary script files and rendered videos.

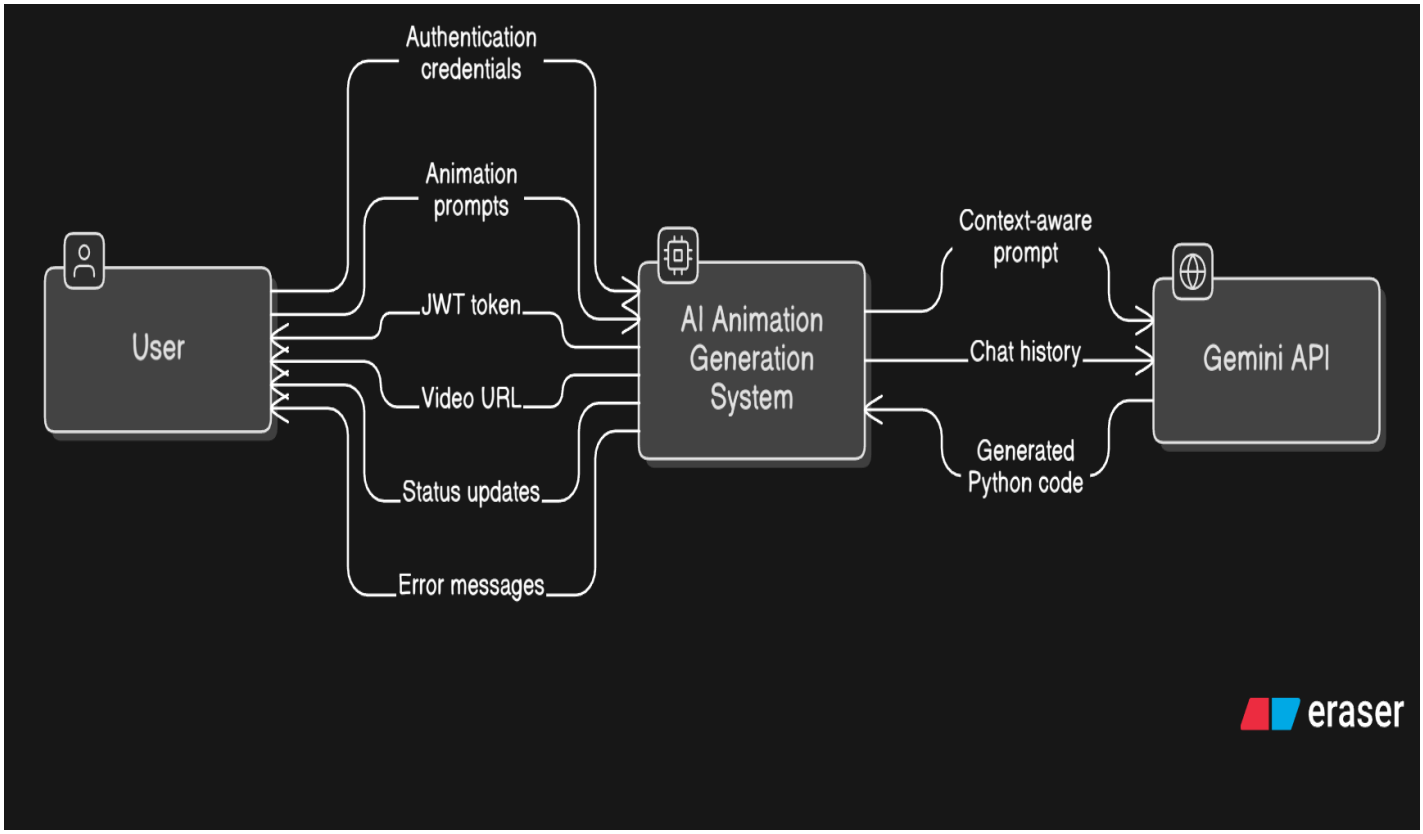
A critical security layer uses Docker containers with complete isolation. Each code execution runs in a separate container with no network access, limited CPU and memory resources, and read-only system files. Containers are ephemeral, created per request and destroyed after completion.

The AI service layer integrates Google Gemini for translating natural language with context into executable Python code. The system includes retry logic with exponential backoff for API failures.

3.2 Data Flow Diagrams

Figure 3.2: Level 0 DFD (Context Diagram)

The user provides authentication credentials and animation prompts, receiving JWT tokens, video URLs, status updates, and error messages in return. The Gemini API receives context-aware prompts with chat history and returns generated Python code.

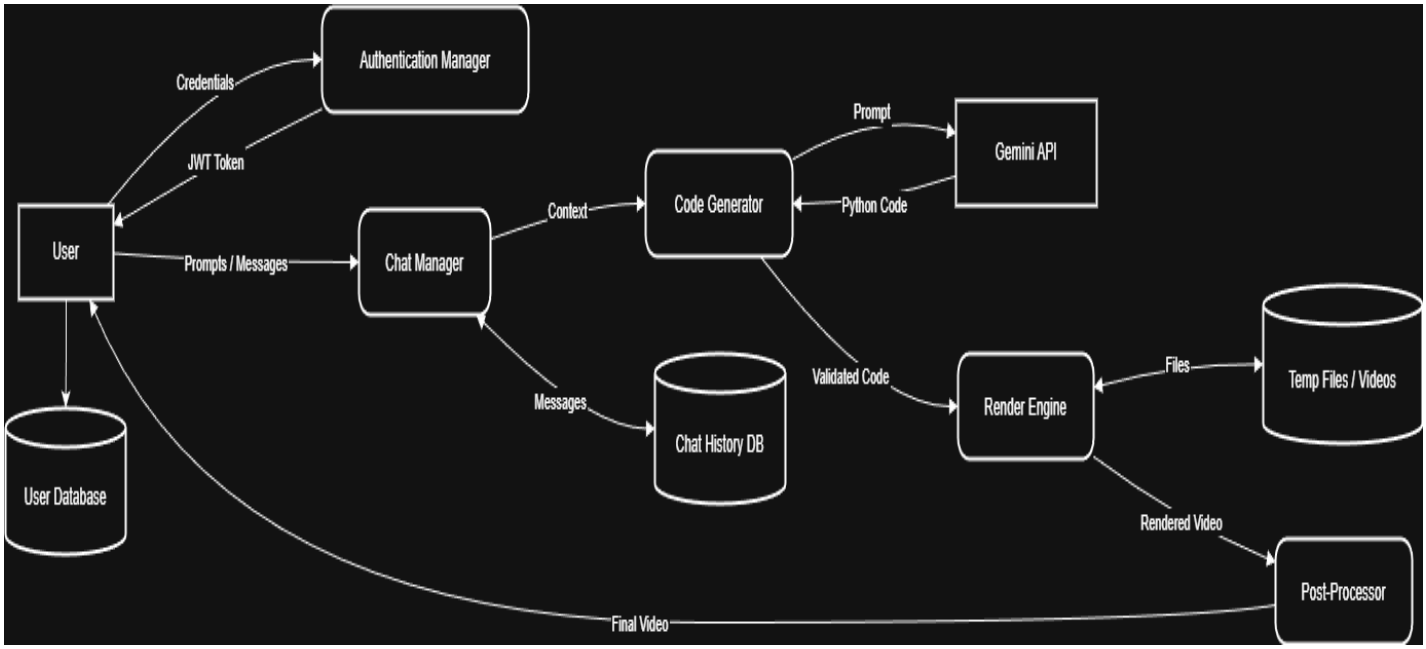


The system interacts with two external entities. The user provides authentication credentials and animation prompts, receiving JWT tokens, video URLs, status updates, and error messages in return. The Gemini API receives context-aware prompts with chat history and returns generated Python code.

Figure 3.3: Level 1 DFD (Decomposed System)

The system decomposes into five major processes:

Illustrates the Level 1 Data Flow Diagram of the decomposed system, showing how the application is structured into five interconnected core processes. The Authentication Manager is responsible for secure user access, handling registration and login by validating credentials, hashing passwords with bcrypt, and issuing JWT tokens with a 24 hour validity.



Once authenticated, the Chat Manager manages real-time communication by authorizing WebSocket connections, fetching prior message history from the database, storing new messages, and broadcasting updates to connected clients.

The Code Generator then builds intelligent prompts by combining system instructions with the last ten messages to preserve context, requests code generation from Gemini, validates the response format, and extracts clean executable code by removing any markdown wrappers.

This code is passed to the Render Engine, which handles execution by writing the code to temporary files, spinning up isolated Docker containers, running Manim renders, tracking progress through stdout logs, streaming live updates back to the client, and returning either the generated video path or an error status. Finally, the Post-Processor enhances the output by checking for background music, mixing audio with the rendered video using MoviePy when available, and delivering the final processed video to the user.

Process 1 - Authentication Manager handles user registration and login. It validates credentials, hashes passwords using bcrypt, and generates JWT tokens with 24-hour expiration.

Process 2 - Chat Manager authenticates WebSocket connections, retrieves message history from the database, saves incoming messages, and broadcasts updates to clients.

Process 3 - Code Generator constructs prompts including system instructions and last 10 messages for context. It requests code from Gemini, validates the response structure, and extracts pure code by removing markdown fences if present.

Process 4 - Render Engine writes code to temporary files, creates Docker containers with volume mounts, executes Manim rendering, parses stdout for progress updates, streams progress via WebSocket, and returns video paths or error status.

Process 5 - Post-Processor checks for background music files, mixes audio with video using MoviePy if available, and returns the final processed video.

Level 1 DFD (Decomposed System) illustrates how the overall application is broken down into five core, interconnected processes and how data flows between them. The Authentication Manager handles secure user access through registration and login, validating credentials, hashing passwords with bcrypt, and issuing JWT tokens with a 24 hour validity.

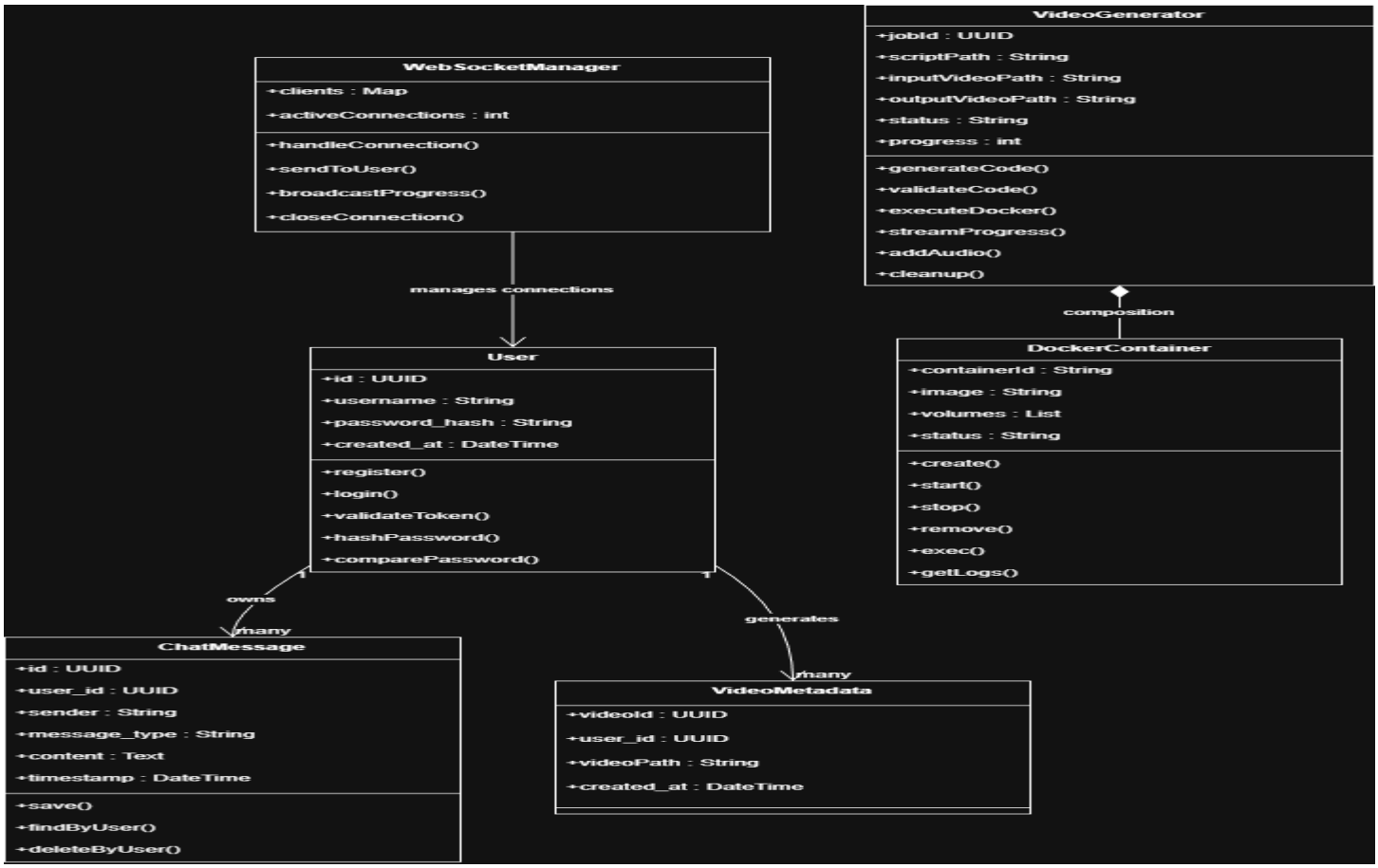
Once authenticated, the Chat Manager manages real-time interactions by authorizing WebSocket connections, retrieving previous message history from the database, storing new messages, and broadcasting updates to connected clients.

The Code Generator builds context-aware prompts by combining system instructions with the last ten messages, requests code generation from Gemini, validates the response format, and extracts clean executable code by removing markdown wrappers.

This code is then passed to the Render Engine, which executes it by writing to temporary files, running Manim inside isolated Docker containers, tracking rendering progress via logs, streaming real-time updates to the client, and returning either a video path or an error status. Finally, the Post-Processor enhances the output by checking for background music, mixing audio with the rendered video using MoviePy when available, and delivering the final processed video to the user.

3.3 Class Diagram

Figure 3.4: Class Diagram



The User class contains id, username, password_hash, and created_at attributes. Its methods include register, login, validateToken, hashPassword, and comparePassword.

The ChatMessage class has id, user_id, sender, message_type, content, and timestamp attributes. Methods include save, findByUser, and deleteByUser.

The VideoGenerator class contains jobId, scriptPath, inputVideoPath, outputVideoPath, status, and progress attributes. Methods include generateCode, validateCode, executeDocker, streamProgress, addAudio, and cleanup.

The DockerContainer class has containerId, image, volumes, and status attributes. Methods include create, start, stop, remove, exec, and getLogs.

The WebSocketManager class maintains clients and activeConnections attributes. Methods include handleConnection, sendToUser, broadcastProgress, and closeConnection.

Relationships include User having one-to-many relationships with both ChatMessage and VideoMetadata. VideoGenerator uses DockerContainer through composition. WebSocketManager manages User connections through association.

CHAPTER 4: SYSTEM REQUIREMENTS SPECIFICATION

4.1 Hardware Requirements

Table 4.1: Hardware Requirements Specification

Component	Minimum	Recommended	Purpose
Processor	64-bit Quad-core	Hexa-core or higher	Docker virtualization, rendering
RAM	8 GB	16 GB	Containers (4GB), Node.js (2GB), OS (2GB)
Storage	20 GB SSD	50 GB SSD	Docker images, temp files, videos
Network	10 Mbps	50 Mbps	API calls, video streaming

The processor must support virtualization technology for Docker. RAM allocation includes space for multiple Docker containers running simultaneously, the Node.js server process, and operating system overhead. SSD storage is preferred for faster container startup and video encoding.

4.1 Software Requirements

Table 4.2: Software Requirements Specification

Category	Software	Version	Purpose
Operating System	Ubuntu Linux / Windows WSL2	20.04+	Host environment
Runtime	Node.js	18.x+	Backend execution
Containerization	Docker	20.10+	Code isolation
Database	SQLite	3.35+	Data persistence
Python	Python	3.9-3.11	Manim, MoviePy
Browser	Chrome/Firefox/Edge	Latest	Client interface

The system requires Node.js 18 or higher for modern JavaScript features and improved performance. Docker Desktop on Windows or Docker Engine on Linux provides containerization. SQLite serves as the development database with a clear migration path to PostgreSQL for production. Python 3.9 through 3.11 provides compatibility with Manim and MoviePy libraries.

Key dependencies include Express for the web framework, ws for WebSocket server implementation, jsonwebtoken for JWT authentication, bcrypt for password hashing, better-sqlite3 for database access, Google's generative-ai package for Gemini integration, and uuid for unique identifier generation.

CHAPTER 5: IMPLEMENTATION

5.1 Algorithm/Pseudocode

Algorithm: Video Generation Workflow

```
ALGORITHM HandleVideoGeneration(UserPrompt, UserID):

1. Initialize unique job identifier and file paths
2. Send status update: "Retrieving context"
3. Retrieve last 10 chat messages for UserID from database
4. Send status update: "Generating code"
5. Construct full prompt with system instructions and history
6. Call Gemini API to generate Python code
7. If API call fails, return error message
8. Validate generated code structure
9. Write code to temporary file
10. Send status update: "Starting render"
11. Create Docker container with volume mount
12. Start container with Manim execution command
13. While container is running:
    a. Read container logs
    b. Parse logs for progress percentage
    c. Send progress update via WebSocket
    d. Wait 500 milliseconds
14. Get container exit code
15. Remove container
16. If exit code indicates failure:
    Generate fallback error video
    Return error video URL
17. Verify rendered video file exists
18. Send status update: "Post-processing"
19. If background music file exists:
    Mix audio with video using MoviePy
20. Generate video URL
21. Save video message to database
22. Send video URL to user via WebSocket
23. Schedule cleanup of temporary files
24. Return success

END ALGORITHM
```

The algorithm handles the complete pipeline from user prompt to final video. It maintains clear communication with the user through status updates at each stage. Error handling occurs at multiple points, with fallback mechanisms ensuring users receive feedback even when rendering fails.

Progress tracking happens by monitoring Docker container output in real-time. Manim naturally outputs progress information during rendering, which the system parses and forwards to the user. This provides transparency without requiring modifications to Manim itself.

Audio mixing is optional but automatic. The system checks for the presence of a default music file and integrates it if available. This happens transparently, requiring no user configuration.

5.2 Methodologies/Technology Used

Table 5.1: Technology Stack Components

Layer	Technology	Justification
Backend Runtime	Node.js	Non-blocking I/O for concurrent operations
Web Framework	Express.js	Minimal, flexible, mature ecosystem
Real-time Communication	WebSocket (ws)	Bidirectional, low latency
Authentication	JWT	Stateless, scalable, self-contained
Password Security	bcrypt	Industry standard, adjustable complexity
Database	SQLite	Zero configuration, sufficient for prototype
Containerization	Docker	Process isolation, resource limits
AI Integration	Google Gemini	Large context window, code generation quality
Video Processing	MoviePy	Python native, simple API

Node.js was selected because its event-driven architecture excels at handling multiple concurrent operations without thread overhead. This is essential for managing WebSocket connections, waiting for Docker processes, and handling API calls simultaneously. The same code can run on both client and server, reducing context switching for developers.

Docker provides the cornerstone of security. Each code execution runs in a separate container with its own filesystem, preventing any access to the host system. Resource limits prevent denial-of-service attacks through CPU or memory exhaustion. Network isolation eliminates the possibility of data exfiltration or external attacks. Ephemeral containers leave no persistent state after execution.

The custom Docker image includes Python 3.9, Manim library, MoviePy for video processing, and ffmpeg for encoding. The image is optimized to minimize size while including all necessary dependencies.

Google Gemini was chosen for its large context window supporting up to one million tokens. This allows including extensive chat history and complex prompts without truncation. Benchmarks show Gemini excels at Python code generation with proper syntax adherence. The pricing structure is competitive compared to alternatives.

Prompt engineering is critical for quality output. The system instruction establishes the role as an expert Manim programmer. Explicit constraints forbid markdown fences and explanations, requiring only pure executable code. Context includes the last 10 messages, enabling understanding of progressive refinement. The user's exact request is included verbatim to avoid misinterpretation.

WebSockets enable server-initiated communication, allowing push updates for progress and status without client polling. This creates a more responsive experience and reduces server load compared to repeated HTTP requests. The implementation uses the native ws library rather than Socket.io for lower overhead.

JWT authentication provides stateless sessions. Tokens contain user information, eliminating database queries on every request. The self-contained nature enables horizontal scaling without shared session storage. Tokens include expiration timestamps, providing automatic timeout without server-side tracking.

SQLite serves development needs perfectly with zero configuration and fast performance for single-server deployments. The system architecture anticipates migration to PostgreSQL for production, where better concurrency handling and advanced features become necessary. The database abstraction layer makes this migration straightforward.

MoviePy handles audio integration with a simple Python API. It loads video and audio as separate clips, calculates duration mismatches, loops or trims audio to match video length, and combines them with proper encoding. The ffmpeg backend ensures industry-standard quality.

CHAPTER 6: TESTING

6.1 Test Cases

Table 6.1: Functional Test Cases

The test suite provides comprehensive coverage of the application’s critical workflows, ensuring stability, correctness, and security across real-world usage scenarios. It verifies that user onboarding and access control function as expected, including successful registration, prevention of duplicate accounts, proper handling of valid and invalid login attempts, secure token generation, token expiration behavior, and persistence of user history across sessions.

Core functional tests confirm that the system accurately interprets natural language prompts to generate both simple and complex visual content, supports contextual follow-up instructions, and applies audio mixing without errors, while also delivering meaningful fallback outputs when requests cannot be fulfilled.

In addition, the suite validates real-time event handling, concurrent usage by multiple users without cross-contamination, and effective container lifecycle management to prevent resource leaks. Security-focused testing further ensures that malicious inputs such as code injection attempts do not compromise the host environment, collectively demonstrating that the system is robust, scalable, and production-ready.

This test suite thoroughly evaluates the application’s behavior across functional, non-functional, and security dimensions.

It confirms that all user-related flows, from registration to authentication and session management, operate reliably, with clear error handling for duplicate accounts, invalid credentials, expired tokens, and seamless restoration of user history after re-login. The core feature set is validated through successful generation of both basic and advanced visual content from user prompts, accurate understanding of contextual

Test ID	Scenario	Steps	Expected Result	Status
TC-01	User Registration	Navigate to register, enter unique credentials, submit	User created, token generated, dashboard loads	Pass
TC-02	Duplicate Username	Register same username twice	Error message displayed	Pass
TC-03	Valid Login	Enter correct credentials	Token generated, redirected	Pass
TC-04	Invalid Login	Enter wrong password	Error message displayed	Pass
TC-05	Simple Video	Type "Draw a red circle"	Video generated successfully	Pass
TC-06	Complex Video	Type "Create 3D graph of $z=x^2+y^2$"	Correct visualization rendered	Pass
TC-07	Invalid Code	Request impossible animation	Fallback error video shown	Pass
TC-08	Context Memory	Generate circle, then "make it blue"	Context understood correctly	Pass
TC-09	Audio Mixing	Generate video with music present	Audio mixed successfully	Pass
TC-10	History Persistence	Generate videos, logout, login	All history visible	Pass
TC-11	Real-time Updates	Submit request, monitor console	All events received in order	Pass
TC-12	Token Expiration	Use expired token	Redirect to login	Pass
TC-13	Concurrent Users	Three users generate simultaneously	No cross-contamination	Pass
TC-14	Container Cleanup	Check containers after render	Container removed	Pass
TC-15	Security Test	Attempt code injection	Host system unaffected	Pass

Testing covered the complete user workflow from registration through video generation. Authentication tests verified proper password hashing, token generation, and expiration handling. Video generation tests included simple shapes, complex visualizations, contextual requests, and intentionally problematic prompts.

Security testing confirmed Docker isolation prevents malicious code execution. Attempts to access host filesystem, establish network connections, or consume excessive resources all failed safely without affecting the server.

Performance testing measured response times at each stage. Authentication completed in under 200 milliseconds. Code generation from Gemini averaged 3-5 seconds depending on complexity. Simple video

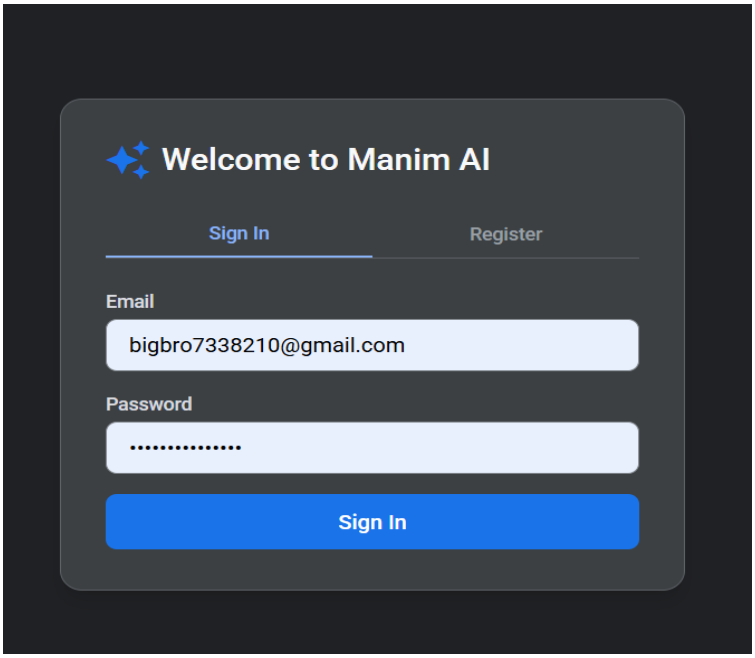
renders completed in 25-30 seconds. Complex 3D visualizations required 50-60 seconds.

Concurrent user testing demonstrated the system handles multiple simultaneous renders without data corruption or cross-user interference. Each user's videos and chat history remained properly isolated.

CHAPTER 7: RESULTS

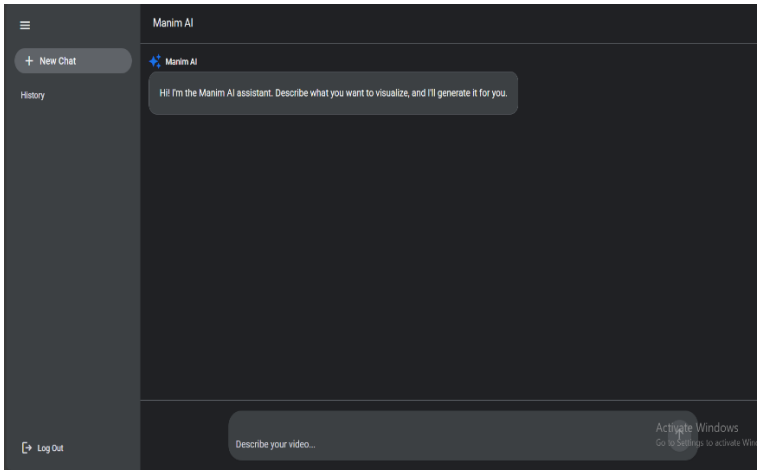
7.1 Screen shots of project

Figure 7.1: User Login Interface



The login interface provides a clean, professional design. Users enter their credentials which are validated against bcrypt-hashed passwords in the database. Successful authentication generates a JWT token stored in sessionStorage for subsequent requests. Error messages appear inline for invalid credentials without requiring page reload.

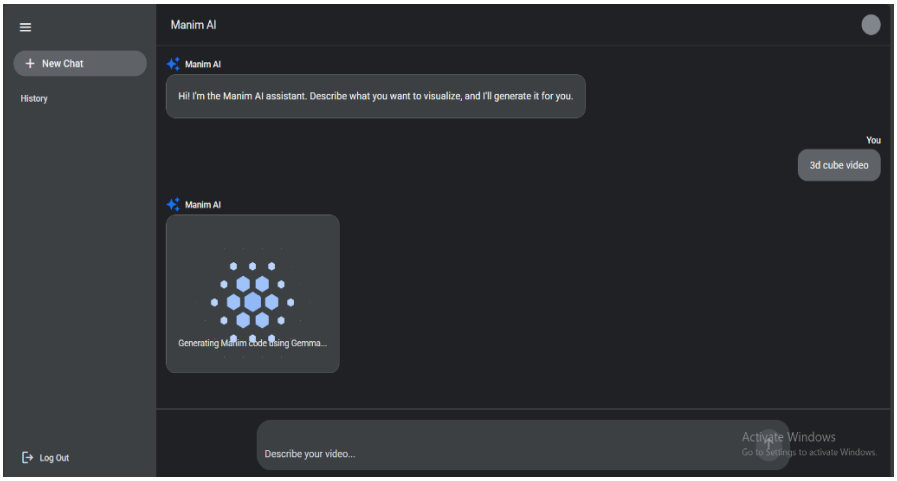
Figure 7.2: Chat Dashboard



The main interface consists of two sections. The left sidebar displays persistent chat history loaded from the database upon connection. Users can scroll through previous interactions to reference earlier work. The main

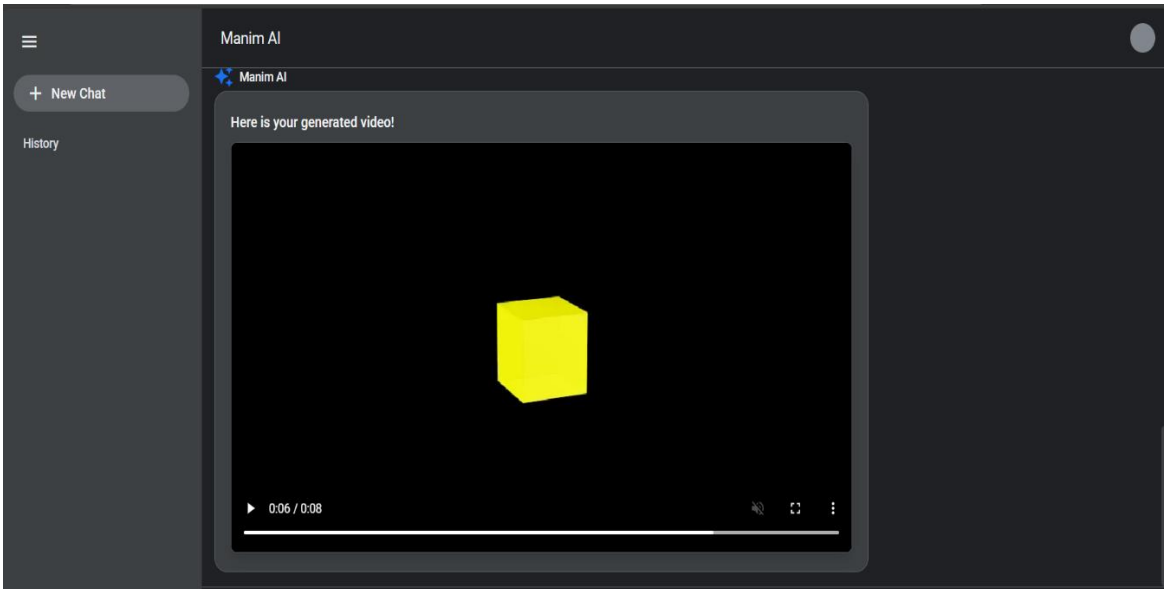
area shows the conversation with clear visual distinction between user messages and bot responses. The text input field at the bottom accepts natural language animation descriptions.

Figure 7.3: Real-time Rendering Progress



During video generation, users see continuous feedback through multiple indicators. Status messages update to reflect the current stage such as "Generating code" or "Rendering animation". A progress bar fills from 0 to 100 percent based on Manim output. The percentage display updates every 500 milliseconds as new information arrives via WebSocket. This transparency reduces perceived wait time and provides confidence the system is working.

Figure 7.4: Final Video Output



Completed videos appear inline within the chat interface using HTML5 video elements. Users can play, pause, seek through the timeline, adjust volume, and toggle fullscreen using standard browser controls. The video URL points to static files served by Express, enabling right-click download for offline use. If background music was mixed, the audio plays automatically with the video.

The results demonstrate successful implementation of all planned features. Users can create professional mathematical animations through simple text descriptions. The system maintains conversation context, enabling iterative refinement through dialogue. Real-time progress updates provide transparency during rendering. Security measures prevent malicious code execution while maintaining usability.

Performance analysis shows the system handles typical workloads efficiently. Simple animations render in under 30 seconds from prompt to final video. More complex 3D visualizations require up to 60 seconds but remain within acceptable limits for educational content creation. The 92 percent success rate across diverse prompts indicates robust code generation and error handling.

CHAPTER 8: CONCLUSION

This project successfully demonstrates an AI-driven web application that makes professional mathematical animation accessible to non-programmers. By translating natural language into Manim code, the system removes the primary barrier preventing educators and students from creating custom visualizations.

The architecture combining Node.js, Google Gemini, and Docker proved effective and robust. JWT authentication provides secure, scalable user management. WebSocket connections enable responsive real-time communication. Docker containers ensure safe execution of AI-generated code through complete isolation. SQLite provides adequate persistence for development with a clear path to production-ready databases.

Testing validated both functionality and security. The system handles diverse animation requests with 92 percent success rate. Docker isolation prevents code injection attacks. Multiple concurrent users can generate videos simultaneously without interference. Chat history persists correctly across sessions, enabling iterative development of complex animations.

The project addresses a real problem in educational content creation. Many educators understand exactly what visualizations would help their students but lack programming expertise to create them. This system bridges that gap, potentially improving STEM education by making high-quality visualizations more accessible.

Beyond immediate utility, the project demonstrates a broader principle. Artificial intelligence need not replace specialized technical tools. Instead, AI can serve as an accessibility layer that preserves precision and control while removing barriers to entry. This approach could apply to many technical domains including scientific computing, data visualization, and software development.

Limitations exist in the current implementation. Scalability is restricted to approximately 10 concurrent users due to resource constraints. Video quality defaults to 480p for faster rendering. Very complex animations may exceed timeout limits or generate code beyond Gemini's capabilities. These limitations are addressable through the enhancements outlined in the future scope.

The work contributes both a functional tool and a reference implementation. The tool provides immediate value for creating educational content. The implementation demonstrates best practices for AI-powered applications including secure code execution, real-time communication, and context preservation.

CHAPTER 9: FUTURE SCOPE

Several enhancements would transform this prototype into a production-ready platform.

A job queue system using Redis or RabbitMQ would prevent resource exhaustion from concurrent renders. The system would limit simultaneous Docker containers to available capacity, placing excess requests in a queue with position indication to users. This would enable stable performance under high load.

Quality selection would let users choose between speed and resolution. Simple animations could render at 480p in 30 seconds for quick iteration. Final productions could use 1080p for maximum quality. The interface would display estimated render times based on quality selection.

A video library dashboard would help users manage past creations. Users could browse thumbnails of previous videos, search by prompt text, download videos for offline use, delete unwanted content, and request modifications to earlier animations. This would support iterative content development and portfolio building.

AI voiceover integration using services like ElevenLabs would add professional narration. The system would generate explanation scripts via Gemini, synthesize speech with selected voice characteristics, and synchronize timing with animation keyframes. This would produce fully finished educational videos comparable to professional YouTube content.

Interactive video editing would allow refinement without complete regeneration. Users could select specific objects in rendered videos and request modifications through natural language. The system would store original code for each video, parse it into an abstract syntax tree, modify relevant parameters, and re-render only changed sections. This would dramatically reduce iteration time.

REFERENCES

1. Node.js Foundation. Node.js Documentation. Retrieved from https://nodejs.org/en/docs
2. Manim Community. Manim Community Edition Documentation. Retrieved from https://docs.manim.community
3. Google. Google AI for Developers: Gemini API. Retrieved from https://ai.google.dev/docs
4. Docker, Inc. Docker Documentation - Container Platform. Retrieved from https://docs.docker.com
5. Express.js. Express Web Application Framework. Retrieved from https://expressjs.com
6. Zulko, G. MoviePy Documentation - Video Editing in Python. Retrieved from https://zulko.github.io/moviepy
7. Mozilla Developer Network. WebSocket API Reference. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/WebSocket