Project Documentation

1. Executive Summary

SHE<Codes/> successfully bridges the gap between gaming and education by creating a scalable, empowering platform designed to teach girls Python fundamentals while exploring the rich contributions of women in STEM history. It is a narrative coding game designed to inspire young women and queer youth to explore careers in computer science and STEM.

The core problem addressed is the lack of programming learning tools that are simultaneously engaging, inclusive, and history-focused.

Problem-Solution

- **The Engagement Gap:** Traditional coding tutorials are often abstract, intimidating, or boring, leading to low retention among young beginners. SHE<Codes/> solves this by using a narrative game format to make learning engaging, creative, and fun.
- **The Gender Gap:** Girls often feel excluded from STEM fields due to a lack of relatable role models and peers. SHE<Codes/> solves this by focusing on the contributions of historical women in STEM, providing inspiring, visible role models. *"You can't be what you can't see"*

Target Audience

The game targets Middle/High School students (11-18 years old) and Adults (18+), with a particular focus on creating an empowering and inclusive learning environment for young women and queer youth.

Solution Approach

The game immerses players on an adventure to repair a time machine, transporting their consciousness to pivotal moments in computing history. Players learn programming concepts by solving interactive coding puzzles presented by historical pioneers like Ada Lovelace, Grace Hopper, Alan Turing, and Katherine Johnson. Challenges progress from typing placeholder code (for beginners) to fill-in-the-blank and Python coding (for advanced levels).

Key Results

The objective is to transform the perception of coding, making it feel empowering, inclusive, and creative, while applying the fundamentals of software engineering and computer science theory. Players earn thematic badges and achievements inspired by the pioneers, fostering motivation and engagement.

The game follows a consistent practice pattern: Story → Guided Example → Challenge → Reflection.

2. Problem & Requirements (O1)

Context, Stakeholders, and Personas

Category	Description
Context	Capstone Project to develop a web adventure game, SHE<Codes/>, that teaches Python programming through stories about computer science pioneers.
Stakeholders	**Alessandra Ortega** (Product Owner), **Bruna Gentil**, **Gabriel Suarez**, **Rajiv Chevannes**, **Farjana Islam Rahin** (Development Team), Instructor (Masoud Sadjadi), STEM Community, and Educators.
Personas	**High School Students (11-18)**: Seek visual, reward-based learning with engaging stories. **Adults (18+)**: Seek self-directed, practical learning relevant to their careers.

Functional Requirements (FR)

The functional requirements detail the necessary features and core gameplay mechanics.

- **Narrative-Driven Core:** The platform **must teach coding concepts through a story** called *"The Digital Time Machine"* featuring historical female pioneers as guides and characters.
- **Difficulty & Progression:** Implement the "The Digital Time Machine" storyline with 3 difficulty levels: Beginner, Intermediate, and Advanced.
- **Scaffolded Learning:**
    - **Beginner:** Allow puzzle solving via drag-and-drop block code (visual blocks).
    - **Intermediate/Advanced:** Must progress to real Python code, including fill-in-the-blank and independent coding challenges.
- **Historical Interactions:** Include interactions with pioneer characters who explain programming concepts and contextualize the challenges within computing history.

Non-Functional Requirements (NFR)

These requirements focus on the quality, constraints, and underlying architecture of the platform.

- **Accessibility & Inclusive Design:**
    - **Comprehensive Accessibility:** Support for screen readers, complete keyboard navigation, diverse language, and representation in characters.
    - **Ethical Considerations:** Address the ethical and privacy implications of software as part of the curriculum.
- **Performance & Scalability:**
    - **Accessibility (Technical):** Must be a web-first application optimized for Chromebooks and low-bandwidth school networks to allow access for all.
    - **Architecture:** Web-first architecture to ensure reliable operation on low-end devices and basic laptops.

Constraints

Content must be age-sensitive, with concepts per challenge to avoid cognitive overload. Feedback must be immediate.

Success Criteria

High retention of Python concepts, high player engagement (measured by application usage, level completion, and hint usage), and a sense of empowerment among the target audience.

Risks

Risk	Mitigation
Excessive use of hints	Reduce player XP (Experience Points) when Tier 2 hints are used.
Device constraints (Performance)	Automatically disable animations on low-end devices.

Prioritized Backlog

1. Game Design Recommendation
2. Technology Recommendation
3. Game Plan
4. Market Research
5. Repo sharing on Github
6. Develop Intro & Start Flow
7. Design Levels Skeleton
8. Mini Game Logic for Levels
9. Connect Backend with Frontend
10. Finish documentation

3. System Overview & Architecture (O2)

Overall Architecture

The SHE<Codes/> application is built on a lightweight, layered architecture using a "Web-First" approach to ensure maximum reach and compatibility with basic hardware devices like Chromebooks.

Core Structure

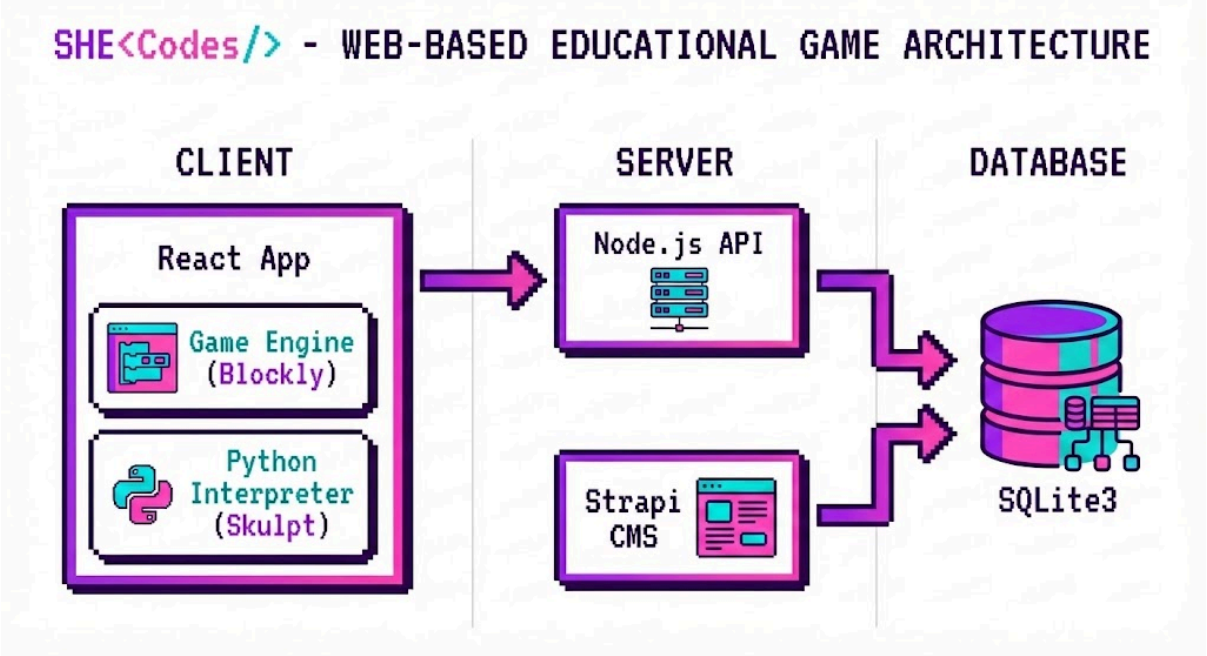
The system follows a monorepo setup for development, hosting the frontend and backend code in one place. This decision was made to simplify collaboration and streamline Continuous Integration (CI) processes.

Backend Architecture

The backend utilizes a generally MVC-like structure (Model-View-Controller, though implemented as Route → Controller → Service → Model) optimized for simplicity and performance on low-end devices. This layering is key to separating I/O concerns from business logic for enhanced testability.

Frontend Architecture

The frontend uses a dual-approach: a simple Scene-per-file HTML/JS setup for quick scene authoring, and a React component-based structure for more complex UI sections. This hybrid approach balances the need for fast, isolated puzzle development with the benefits of modern component-based design.



Key Technologies, Frameworks, and Services

Component	Technology	Description
Frontend	**React** **TypeScript**	Industry standard for interactive, component-based UIs, with strong typing to reduce errors.
Styling	**Tailwind CSS**	CSS framework for quickly building responsive and accessible layouts.
Code Editor	**Blockly**	Visual block-based coding tool for beginner-level challenges.
Interpreter	**Skulpt**	Lightweight Python-in-JavaScript interpreter, allowing students to run Python code directly in their browser.
Backend	**Node.js** **Express**	Scalable and widely compatible solution for handling API requests (saving progress, retrieving content).
Database	**SQLite**	Robust relational database for storing user information, progress, achievements, and story content.

Security/Auth	**JWT + bcrypt CORS**	Stateless authentication using JSON Web Tokens (JWT) and secure password hashing.
Build Tools	**Vite**	Modern build tool and development server that provides instant hot module replacement (HMR) and lightning fast builds for React applications

Development Practices

- **Continuous Integration (CI):** GitHub Action**s** are used to automatically run tests, code style checks, and type safety checks before code merges.
- **End-to-End Testing (E2E):** Playwright is used to ensure key flows work on browsers.

API Surface and Data Storage Overview

- **API Surface:** The architecture is compatible with these interfaces via the Node.js/Express backend, although the detailed specification is not provided in the current documentation.
    - Protocol: REST (HTTP/JSON)
    - Headers: CORS-enabled
    - Authentication: JWT Bearer tokens
    - Validation: Server-side (express-validator) + client-side (implicit via scene logic)
- **Data/Storage Overview:** The SQLite database stores persistent data (user profiles, progress, achievements, story content, and levels).
    - Primary DB: SQLite (file-based)
    - Tables: users, user_progress, levels, code_submissions
    - Read/Write: Synchronous + Promise-based async in services layer

4. Discipline-Specific Depth (O6)

Algorithms & Data Structures

Component	Algorithm	Complexity	Risk
Progress Aggregation	COUNT DISTINCT aggregation	O(k) per user (k = progress rows)	Full table scan; scales poorly with user history
Code Validation	Substring matching loop	O(n × L) (n = keywords, L = code length)	Brittle; doesn't check semantic structure
Code Submission	Single-row INSERT	O(1) write	SQLite write serialization bottleneck
Query Lookups	Index-based PK/FK lookups	O(1) avg (if indexed)	Missing indexes → full table scans

Architecture & Patterns

Rationale for Decomposition

- **Routes:** Responsible *only* for mapping the HTTP method/path to the correct Controller.
- **Controllers:** Handle I/O and HTTP concerns (receiving and validating input, orchestrating calls to services, and returning the HTTP response).
- **Services:** Contain all business logic and database operations. Decoupled from HTTP, easy to unit test.
- **Models/DB:** Manages the database schema, connection, and raw SQL/ORM interactions.

Concurrency Model

The system runs on Node.js, leveraging its single-threaded event loop for concurrency.

- **Behavior:** The non-blocking I/O model is highly efficient for handling many simultaneous readers (low-latency requests).
- **Constraint:** The use of SQLite is the primary bottleneck. As a file-based DB with a single-writer lock, concurrent write operations are serialized, leading to increased latency under write bursts.
- **CPU Constraint:** Any heavy CPU task (like parsing code or execution) will block the event loop unless explicitly offloaded.

State Management

- **Backend State:** The persistent data resides in SQLite, making the server the source-of-truth.
- **Frontend State:** State is per-scene, utilizing local variables and the DOM. There is no global store across scenes. This promotes isolation but can complicate cross-scene interactions.
- **Authentication State:** Authentication is stateless using JWTs, which the server verifies against the database for user information.

Engineering Evidence

Requirement	**Purpose**
Response Time	Quantifies performance; verifies that most requests return within an acceptable time.
Bottleneck Identification	Verifies the need for and effectiveness of moving CPU-heavy tasks to worker threads.
Load Handling	Confirms the safe traffic limit before the SQLite bottleneck (single-writer lock) severely degrades performance.
Low-End Performance	Validates the "Web-First" and low-bandwidth optimization goals.

5. Implementation Notes (O2)

Repository structure

```
server/
```

```
├── src/
│   ├── controllers/              # Request handlers
│   │   ├── userController.js         # User operations
│   │   ├── progressController.js     # Progress tracking
│   │   ├── levelController.js        # Level management
│   │   └── codeController.js         # Code submissions
│   ├── services/                 # Business logic
│   │   ├── userService.js            # User management
│   │   ├── progressService.js        # Progress logic
│   │   ├── levelService.js           # Level operations
│   │   └── codeService.js            # Code validation
│   ├── models/                   # Database models
│   │   └── database.js               # SQLite setup & initialization
│   ├── middleware/               # Express middleware
│   │   ├── authMiddleware.js         # JWT authentication
│   │   └── errorHandler.js           # Error handling
│   ├── routes/                   # API routes
│   │   ├── userRoutes.js             # User endpoints
│   │   ├── progressRoutes.js         # Progress endpoints
│   │   ├── levelRoutes.js            # Level endpoints
│   │   └── codeRoutes.js             # Code endpoints
│   ├── utils/                    # Utility functions
│   │   ├── validators.js             # Data validation
│   │   └── helpers.js                # Helper functions
│   └── config/                   # Configuration
│       └── constants.js              # App constants
├── db/                           # Database files
│   └── shecodes.db                   # SQLite database
├── app.js                  # Express app setup
├── package.json            # Dependencies
├── .env                    # Environment variables
└── README.md               # Documentation
```

Notable patterns

- **MVC-like layering:** Routes → Controllers → Services → Models (keeps HTTP logic separate from business logic).
- **Repository Pattern:** Services encapsulate all database logic and business rules.
- **Singleton DB Connector:** A single instance of the SQLite database connection is returned to manage the connection state centrally.
- **Per-scene Frontend Approach:** Isolation and faster authoring by self-contained Per-scene HTML/JS files, alongside React + Vite and the puzzle engines Blockly/Skulpt.
- **Schema & seed on startup:** Using initializeDatabase() creates tables and seeds a test user for dev.

Tradeoffs

Trade-off	Benefit	Implication

Simplicity vs. Scalability	SQLite + Raw SQL: Simple, minimal dependencies.	SQLite: Imposes a single-writer lock, causing write bursts to queue up and increase latency, making it unsuitable for multi-instance scaling.
Isolation vs. Duplication	Allows designers to edit scenes quickly and easily.	Duplication: Lacks a single client state store, increasing maintenance costs and risking inconsistent logic across scenes.
Raw SQL vs. ORM	Raw SQL: Explicit, minimal overhead, clear queries.	Lacks Features: Missing features like migrations, typed models, and the higher-level safety/abstraction an ORM provides.
Observability	Minimal dependencies.	Minimal Logging: Harder to diagnose performance issues in production due to lack of metrics, structured logging, and query profiling.

6. Testing & Evaluation (O3)

Strategy

The project employs a multi-layered testing strategy combining code-level, user-flow, and continuous quality checks.

Testing Type	**Description**
Unit Testing	Validated individual game components to ensure that smaller, critical pieces of the logic function correctly. An example given is the validation of orbit calculation logic to ensure accurate feedback on user code.
E2E Testing	Utilized Playwright to verify critical user flows. This ensures the most important user paths through the application are functional and integrated correctly.
Flows Verified	The specific critical user flow verified is: Login → Start Challenge → Submit Code → Save Progress.

Coverage, Performance, and Reliability Testing

- **Coverage/Quality Checks:** The CI/CD Pipeline implemented GitHub Actions for automated linting and type-safety checks. These checks are run before every merge to enforce code quality standards and catch errors early.
- **Reliability:** The E2E testing of critical user flows confirms the system's ability to maintain core functionality under normal operation.
- **Performance:** There is no explicit detail about dedicated performance or load testing, but the system is designed to be a web-first application optimized for Chromebooks and

low-bandwidth school networks.

7. Security, Privacy, Accessibility & Compliance (O5)

Societal and Ethical Impact

The project's foundational mission is to address the gender gap in STEM and ensure equitable access to quality coding education.

- **Representation and Inclusion:** The videogame features diverse role models and uses historical accuracy to showcase pioneers' contributions while acknowledging their challenges. This approach actively cultivates an empowering and inclusive learning environment for young women and queer youth, directly combating systemic underrepresentation.
- **Accessibility First:** SHE<Codes/> Web-First architecture is optimized for Chromebooks and low-bandwidth networks, ensuring consistent, reliable access across diverse economic and institutional settings.
- **Ethical Curriculum:** The curriculum is designed to be comprehensive, ensuring that as players master real Python syntax, they are also educated on the ethical and privacy implications of software, fostering responsible future engineers. The game promotes a Growth Mindset through positive feedback and avoidance of punitive failure mechanics.

Security and Privacy

The system employs standard, professional security measures to protect user data, establishing a secure foundation for compliance.

- **Secure Authentication:** User data is protected via bcryptjs for password hashing and JSON Web Tokens for efficient, stateless authentication.
- **SQL Injection Prevention:** All database interactions utilize Raw SQL with parameterized placeholders, a proven industry standard for preventing critical injection vulnerabilities.
- **Privacy by Design:** The backend focuses on Data Minimization, concentrating only on persistent, canonical data necessary for progress tracking.

Accessibility and Technical Inclusion

- **Technical Reach:** The JSON-based Backend API ensures machine-readability, enabling alternative client interfaces. Furthermore, the Blockly visual coding editor provides a more accessible entry point for kinesthetic and visual learners.

8. Deployment & Operations

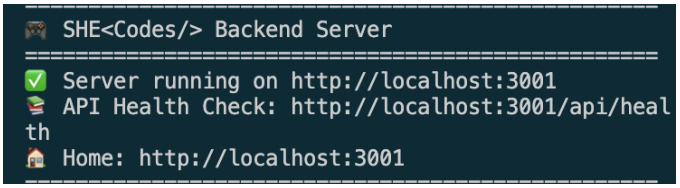
The current version of SHE<Codes/> is deployed as a local web application on a developer machine. The system has two parts: a Node/Express backend with a SQLite database (shecodes.db) and a Vite/React frontend. To run the system, the user clones the GitHub repository, installs Node dependencies in both the /server and /client folders, then starts the backend and frontend using the provided npm scripts. The backend exposes a health-check endpoint (/api/health) and connects to

the SQLite file in /server/db. The game is then accessed through a web browser at http://localhost:5173.

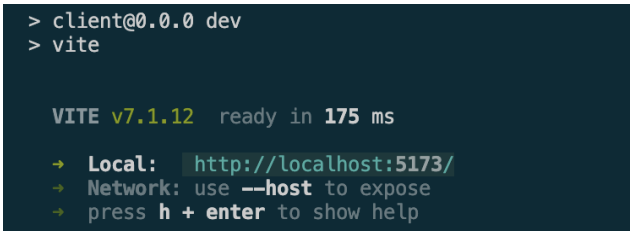
Detailed, step by step deployment and operation instructions (including prerequisites, exact commands, and how to shut down or restart the system) are provided in Appendix A: Deployment & Operations Guide.

Step by step

git clone <git@github.com:Brunang/Python_Game.git>
cd Python_Game
cd ../client
npm install
cd server
npm run dev
You should see:



cd client
npm run dev
You should see:



The game is then accessed through a web browser at http://localhost:5173

9. Limitations & Future Work

Known Limitations and Technical Debt

The current architecture prioritized rapid development but resulted in several key limitations that constrain scalability, quality, and compliance:

- **Scalability & DB Bottleneck:** The SQLite single-writer lock limits write throughput, preventing horizontal scaling. Additionally, the lack of a migrations framework makes schema changes risky, and unbounded table growth for submissions will eventually slow down aggregation queries.

- **Frontend Architecture:** The per-scene, DOM-based state creates logic duplication across files and prevents consistent cross-scene user experiences.
- **Security & Compliance Gaps:** Critical security risks include a lack of rate limiting and input validation on key endpoints. The system has yet to comply with CCPA as it lacks endpoints for data export/deletion.
- **Accessibility Debt:** Frontend scenes need WCAG 2.1 AA elements (ARIA labels, keyboard navigation, color contrast), excluding screen reader and keyboard-only users.

Shortcuts and Workarounds

Shortcut	Rationale	Cost (Long-Term Fix)
SQLite DB	Simple, file-based, no setup required.	Single-writer bottleneck will require migration to Postgres for scaling.
Raw SQL + Callbacks	Minimal dependencies; quick to write.	Must migrate to ORM/Query Builder (Prisma).
Per-scene HTML	Fast scene authoring; isolated logic.	Logic duplication, need to migrate to a single React SPA with shared state.
No Migrations	Fastest path for project duration.	Manual schema changes. High risk of environment inconsistency.

Prioritized Roadmap for Future Iterations

Phase	Priority	Key Roadmap Items
Phase 1: Stabilize and Secure	Critical	Implement DB Indexes, Rate Limiting to prevent DoS, Input Validation, and basic Request Logging.
Phase 2: Quality and Compliance	High	Implement a DB Migration Framework; add CCPA compliance endpoints; and begin the WCAG 2.1 AA accessibility audit and fixes.
Phase 3: Architecture and UX	Medium	Replace brittle code validation with AST checks, centralize frontend state.
Phase 4: Scale and Features	Nice-to-Have	Migrate to Postgres, add WebSocket real-time sync, and implement PWA/offline support.

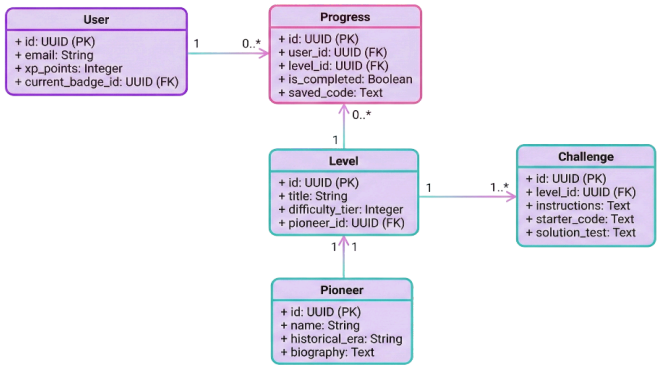
Appendices (Evidence & How-To)

A. Installation Guide

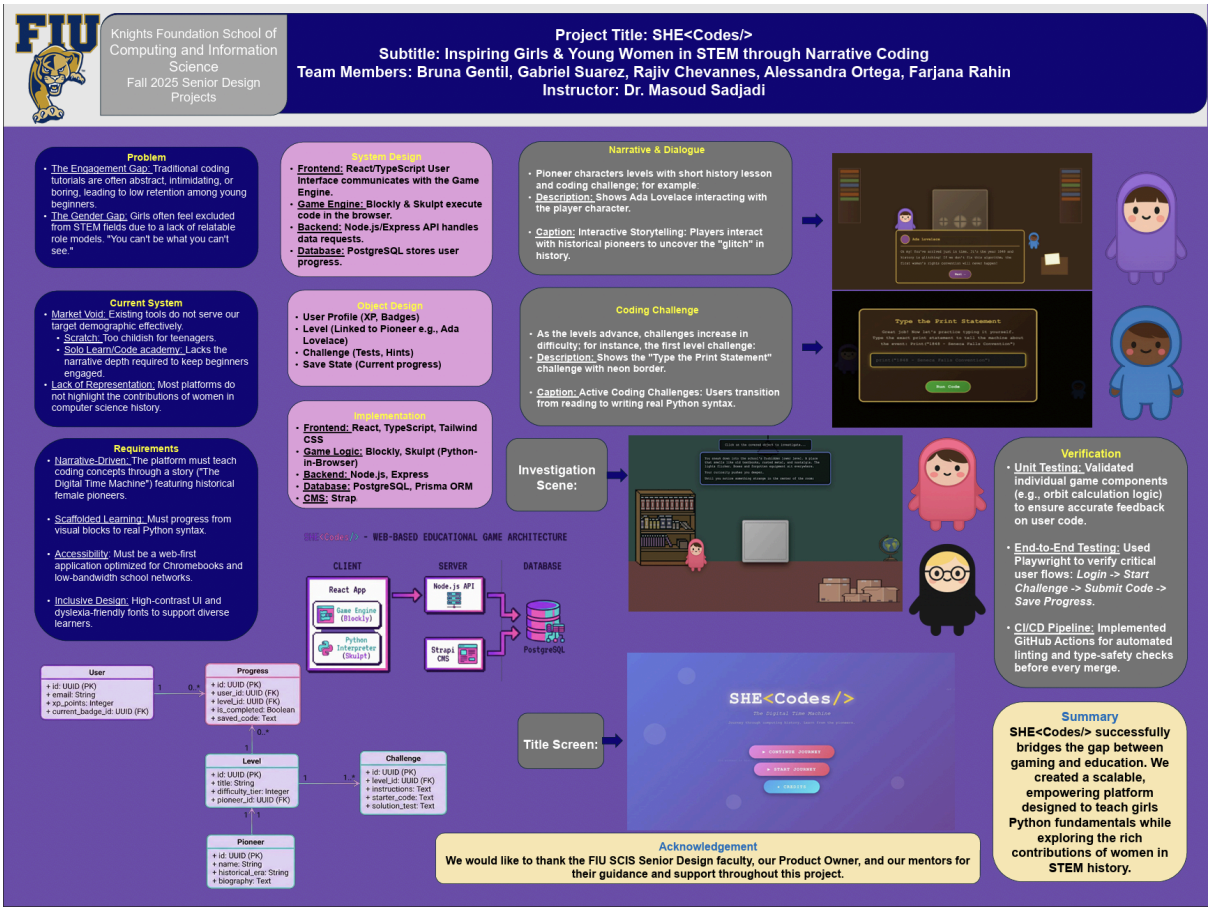
From the README.md file located inside the server file:

````
## Setup

### 1. Install Dependencies

```bash
cd server
npm install
```

### 2. Environment Variables

The `.env` file is already set up with defaults:
- `PORT=5000`
- `NODE_ENV=development`
- `JWT_SECRET=your_jwt_secret_key_change_in_production`
- `DATABASE_PATH=./db/shecodes.db`

### 3. Run the Server

```bash
# Development mode (with auto-reload)
npm run dev

# Production mode
npm start
```

The server will run on `http://localhost:5000`
````

B. API Reference



C. Poster (36"×48" horizontal), Slides, and Video Links (Intro, Demo)



https://fiudit-my.sharepoint.com/:p:/g/personal/frahi003_fiu_edu/IQBnh6nxqzEEQZque1s0V9TiAaq3h9xXrD5bCEtmtZma-Cc?e=9OPTrj

SHE<Codes/> Presentation

https://youtu.be/hoswx01KX30

https://youtu.be/Kp0kivIYBeY?si=UjhhOjmgyNQSlOKc

D. Scrum Evidence Index (links to Sprint Planning, Dailys, Reviews, Retros)

Capstone 2