Summer Internship Report

on

OpenHW Studio: Expanding the Arduino Simulator with New Components and Fixes

Submitted by

Sagar Seth

Jaypee University of Engineering & Technology

Under the Guidance of

Prof. Prabhu Ramachandran

Principal Investigator, FOSSEE Project

Department of Aerospace Engineering,

Indian Institute of Technology Bombay

June 2026

Acknowledgement

I would like to express my deepest gratitude to Prof. Prabhu Ramachandran for providing me with the opportunity to be a part of the FOSSEE Internship Program and for supporting open-source engineering tool development at IIT Bombay. His vision and leadership have inspired students and researchers to actively contribute to the open-source ecosystem.

I would also like to sincerely thank Prof. Rajesh Kushalkar, Senior Project Manager, Open Source Hardware (OSHW), FOSSEE, IIT Bombay, for his valuable guidance, encouragement, and continuous support throughout the internship. His mentorship greatly contributed to my learning and professional growth.

My heartfelt appreciation goes to my mentors, Mr. Pratik Bhosale, Project Research Associate, and Mr. Pratik Nemane, Project Research Assistant, for their constant technical guidance, constructive feedback, and encouragement throughout the project. Their insights played a key role in refining ideas, overcoming challenges, and ensuring the successful completion of the tasks assigned to me.

I would also like to thank my fellow interns and colleagues at OpenHW Studio for their support, collaboration, and constructive discussions throughout the internship. Their motivation and teamwork created a positive learning environment and contributed significantly to my overall experience.

Finally, I extend my sincere gratitude to everyone associated with the FOSSEE Project, IIT Bombay, for fostering an environment of innovation, collaboration, and continuous learning. This internship has been an invaluable experience that strengthened my technical skills and motivated me to continue contributing to the open-source community.

Sagar Seth

Declaration

This internship proved to be a highly rewarding educational journey, granting me the chance to support the creation of virtual hardware modules while integrating functional improvements and debugging within OpenHW Studio. The experience offered significant exposure to professional open-source software practices, micro-controller simulation environments, and integrated development processes. The technical competencies and hands-on insights developed throughout this period will serve as a vital foundation for my upcoming academic projects and career objectives.

I wish to further extend my sincere appreciation to the members of the FOSSEE Team for their seamless coordination, technical mentorship, and unwavering encouragement during the program. Their collaborative spirit and proactive assistance fostered a productive workspace, proving instrumental in the effective delivery of all project milestones and responsibilities.

Sagar Seth

Contents

Executive Summary

This report documents all development work carried out across the OpenHW Studio ecosystem during a six-month internship period. The work spans five critical repositories: the Backend (compilation pipeline, simulation infrastructure, classroom management), the Frontend (gamification, guided learning, teacher tools, live collaboration), the Emulator (hardware component library, circuit validation, WiFi emulation), the Examples repository (guided project content), and the Documentation portal.

Scope of Work

The primary objective was to transform OpenHW Studio from a basic Arduino simulator into a comprehensive educational platform capable of supporting structured learning, classroom management, automated assessment, and real-time collaboration. This required coordinated changes across all five repositories to ensure the frontend, backend, emulator, and content worked together as a cohesive system.

3.1.1 Backend Development

The backend was restructured from a single Express server into a multi-service architecture supporting four distinct hardware compilation targets — Arduino Uno (AVR), Raspberry Pi Pico (RP2040), ESP32 (Xtensa), and STM32 (ARM Cortex-M3). A QEMU-based simulation pipeline was built for ESP32, mirroring a Renode-based pipeline for STM32. A hot pool manager pre-warms virtual machines for near-instant simulation startup. The library management system was split into permanent and cached tiers with automatic eviction. A complete classroom management system was implemented with assignment creation, submission grading, and live session orchestration. The authentication system was extended with Google OAuth, password reset flows, and teacher-initiated student registration.

3.1.2 Frontend Development

The frontend received the most visible transformations. A complete adventure-based gamification system was built with visual world maps, project progression paths, XP milestones, badge rewards, and component unlocks. A four-step guided learning pipeline moves students through theory (flip cards), quizzes (multi-question assessment), component rewards, and finally the simulator. Teachers gained a full project bank with a content editor (1,400+ lines) supporting theory cards, quiz questions, reward components, and auto-generated assessment criteria from circuit PNGs. A real-time live classroom system broadcasts the teacher's simulator workspace to all connected students via WebSocket with delta-based synchronization. Community features allow users to publish, discover, and share circuits. Performance optimizations include lazy loading of the guide page and a 79% reduction in guide data size.

3.1.3 Emulator Development

The emulator component library grew from a basic set to over 70 components with 50+ new additions spanning environmental sensors, displays, motor control, input devices, audio, wireless modules, power management, digital logic, and discrete components. The most technically complex addition was the Pico W WiFi emulation stack — a full behavioral model of the Broadcom CYW43439 chip on the gSPI bus, accompanied by a complete L2–L7 userspace network protocol stack (ARP, DHCP, DNS, ICMP, TCP NAT, UDP NAT). The circuit validation engine enforces 22 physics-based rules using graph traversal. A BaseComponent telemetry

system provides automatic instrumentation for all components with pin tracking, I/O throughput monitoring, and resistor-aware voltage propagation.

3.1.4 Examples and Documentation

The examples repository grew from 11 to 41 guided projects covering sensors, actuators, displays, IoT, communication protocols, and multi-platform support across Arduino Uno, Nano, Mega, Pico, and ESP32. A consistent three-file pattern (guide.json, evaluation.json, code.ino) was established. The documentation was consolidated into a professional VitePress portal with 74 documentation files (15,900+ lines) covering architecture, grading engine, classroom system, telemetry, deployment guides, and AI prompts.

Frontend

Gamification / Adventure System

4.1.1 Architecture and Design

The most substantial frontend addition is a complete adventure-based learning journey built around a visual `AdventureMapPage`. This page displays themed "Worlds" — Circuit Basics, Signal Control, Machines & Sensors — with projects laid out along a winding path. Each world shows lock/unlock status based on XP thresholds, project nodes along the path, and per-world progress bars. The system supports class-based adventures via a `classId` parameter that links adventure content to a teacher's classroom curriculum.

The gamification state manager was significantly expanded to track and reward student progress across multiple dimensions. XP milestones are awarded at 5, 20, and 50 components placed; 10 and 50 wires drawn; and 1 and 10 simulations run. A level-up notification system displays celebratory toasts when thresholds are crossed. Badges are automatically granted and persisted to MongoDB via `unlockService`. A 25% XP re-submission bonus encourages students to revisit and improve their work.

4.1.2 Technical Challenges

One key challenge was preventing API spam from rapid XP-triggering actions. When a student rapidly places components, each placement could trigger an API call, overwhelming the server. A debounced saving mechanism was implemented: XP updates are batched in memory and flushed to the server only after 2 seconds of inactivity. This reduced API calls by approximately 90% during rapid editing sessions.

Another challenge was the wildcard unlock system. Teachers needed the ability to unlock all components for a student at once (e.g., after a student completes a major milestone). The system supports a special `"*"` wildcard token in the unlock payload that the backend interprets as unlocking every component in the library. The frontend checks for this token and updates the local unlock state to reflect all components as available.

The `GamificationGuidePanel` side panel was another challenge — it needed to show component lock status with greyed-out icons for locked parts, wiring guides for the current project, code concepts, and Arduino API references. This required a three-tab interface (Components, Wiring, Code) that dynamically renders content based on the current project's data. The panel fetches component unlock status from the gamification state manager and applies CSS filters to locked component icons, making them visually distinct while preserving their shape for recognition.

Project Learning Flow

4.2.1 Four-Step Pipeline

The guided learning pipeline consists of four sequential steps: Theory, Quiz, Unlock, and Simulate. Each step must be completed before the next becomes available, ensuring students build foundational knowledge before hands-on practice.

ProjectTheoryPage

The theory step uses a flip-card based reading experience. Each card contains an emoji, front title, simple explanation, detail text, and fun fact. The design was inspired by educational platforms

like Duolingo, using bite-sized information chunks to maintain engagement. A progress tracker shows how many cards the student has read versus the total. When all cards are read, the step auto-completes and the quiz step unlocks. The page supports both built-in projects (from a local data store) and custom class-based content (fetched from the teacher's project bank).

ProjectQuizPage

The quiz interface presents multiple questions with A/B/C/D options. Progress dots show completion status, and a 70% passing threshold is enforced. Quiz results are posted to the backend via `adventureService.js`, which validates answers server-side to prevent tampering. On passing, the quiz step is marked complete and the unlock step becomes available. The challenge was ensuring quiz data integrity — since the frontend renders the questions, a technically adept student could inspect the correct answers from the JavaScript source. The backend performs independent answer validation, and the frontend only stores the user's selected answers, never the correct ones.

ProjectComponentUnlockPage

Upon passing the quiz, students claim component rewards. Individual unlock buttons per component provide a satisfying click-to-claim interaction, while an "Unlock All" bulk action expedites the process for students who have already earned the right. Unlocks persist to both class-scoped and global unlock records. The visual feedback includes a brief animation of the component appearing in the student's inventory.

4.2.2 Serial Monitor in ProjectGuidePage

The `ProjectGuidePage` includes a full serial monitor popup that enables users to send serial data to the embedded simulation iframe and view serial output in real time. It supports configurable baud rate, line endings (LF, CR, CR+LF, none), auto-scroll, and pause controls — all rendered inside a draggable popup overlay with a `SerialTabBar`, `SerialOutputPane`, and `SerialSendRow`.

The key challenge was that the simulator runs inside an `<iframe>` while the serial monitor UI lives in the parent React tree. A `postMessage()` bridge relays serial data between the two contexts. The serial output pane uses a virtualized list with `scrollTop` anchoring and a `ResizeObserver` on the content pane to keep the scroll position at the bottom when auto-scroll is enabled, preventing jank during rapid data arrival.

The draggable popup uses pointer events rather than mouse events to support both mouse and touch interactions.

4.2.3 Performance Optimization

The `ProjectGuidePage` was one of the heaviest components in the application. Initial analysis showed it pulled in the entire simulator library, serial monitor code, and the full guided projects dataset on every navigation. Load times exceeded 8 seconds on 3G connections.

The solution was threefold. First, the page was converted to a `React.lazy()` import, code-splitting it out of the main bundle. This means the landing page and dashboard render instantly, and the guide page loads only when the user navigates to it. Second, the `guidedProjects.json` file (151 KB) was analyzed and found to contain 120 KB of unused Arduino schema definitions embedded as part of the project metadata. These were stripped to create `guideProjectsIndex.json` (31 KB), a flat index of 39 projects containing only titles, descriptions, board types, and difficulty levels. The index is eagerly imported inside the

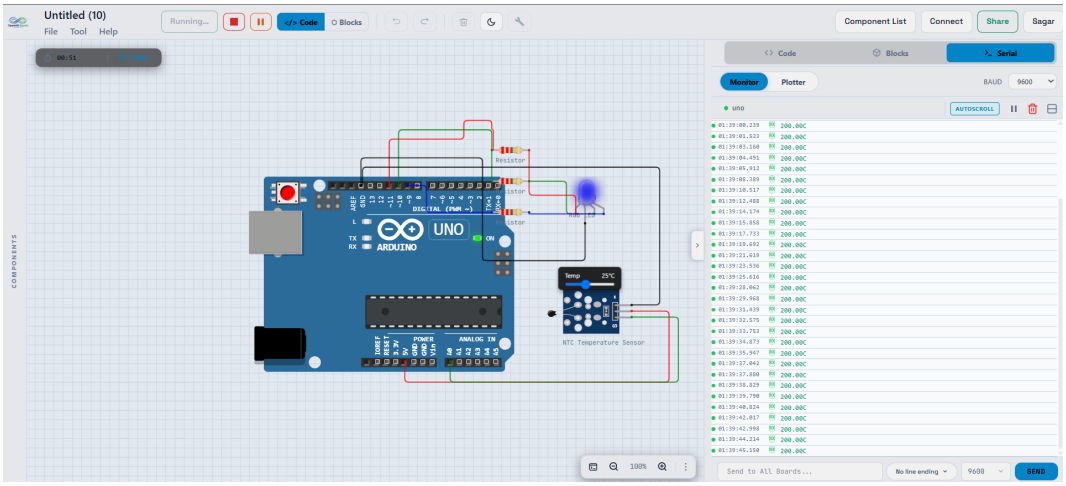

Figure 4.1: Guided project page with serial monitor popup open

lazy chunk so the full project schema is never loaded on the guide page. Third, a `prefetch` link and `fetchpriority="high"` attribute were added to the simulation iframe to prioritize its loading, ensuring the simulator is ready by the time the student reaches it.

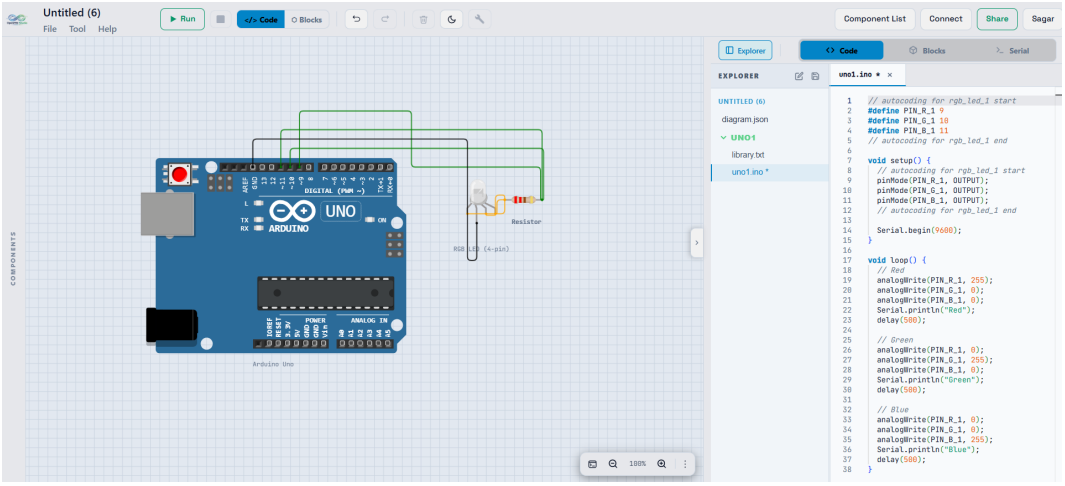

Figure 4.2: Simulation page of the Blink LED project

Teacher Project Bank System

4.3.1 Project Bank Page

The `TeacherProjectBankPage` provides a full UI for managing project templates. It supports two views: “My Projects” (owned by the teacher) and “Shared Projects” (published to all teachers). Board-type filtering covers Arduino Uno, Pico, ESP32, and more. Search by title uses a debounced input that filters the project list in real time. Projects are displayed in a drag-and-drop card layout with delete and “Open to edit” actions.

4.3.2 TeacherProjectContentEditor

The content editor is an extensive 1,400-line component with four tabs:

Theory Tab

A CRUD interface for flip cards with emoji picker (using a Unicode emoji grid), front text, simple explanation, detail text, and fun fact fields. Cards can be added, reordered via drag-and-drop, and deleted. The editor validates that at least one card exists before saving.

Quiz Tab

A CRUD interface for quiz questions with question text, four options (A/B/C/D), and correct answer selection. The editor enforces that exactly one correct answer is selected per question and that at least one question exists.

Rewards Tab

A component picker for unlockable components with search bar, category filters (sensors, actuators, displays, etc.), and a visual component grid showing icons and names. Teachers select which components students unlock upon completing the project.

Assessment Tab

The most technically complex tab. It auto-generates evaluation criteria from uploaded reference circuit PNGs by extracting components, connections, and code functions. The PNG parsing uses `extractProjectMetaFromPng()` from `projectCompilerUtils.js` which reads metadata embedded in the PNG by the simulator's save function. The extracted data populates fields for wiring accuracy weights, code functionality requirements, and component presence checks. Teachers can adjust passing thresholds and save to both class adventures and the project bank.

4.3.3 Auto-Grading Pipeline

The full auto-grading pipeline consists of several coordinated components:

The `TeacherGradingPanel` generates a reference binary key from the teacher's correct circuit by hashing the circuit's normalized component list, connection map, and code into a compact string. This key is stored alongside the assignment.

The `StudentGradingPanel` runs a Web Worker (`grading-engine.worker.ts`) that extracts the student's circuit metadata from their submission PNG. The worker compares component placement, wiring connections, and code functionality against the reference key. Component matching uses fuzzy name matching to handle renamed components.

An optional AI semantic audit via `ai-audit-final.worker.ts` provides natural language feedback on the student's circuit design. This can be disabled via the `VITE_DISABLE_AI_AUDIT` environment variable.

The `GradingPage` standalone tool provides a detailed pass/fail visualization with temporal telemetry comparison, allowing teachers to see how a student's circuit evolved over time.

A `ProjectBankModal` allows selecting projects from the bank when configuring class adventures, providing a searchable list of available templates.

Live Classroom Sessions

4.4.1 Real-Time Broadcasting

A real-time live session system allows teachers to broadcast their simulator workspace to all connected students via WebSocket. Teachers create a session from the class detail page, which

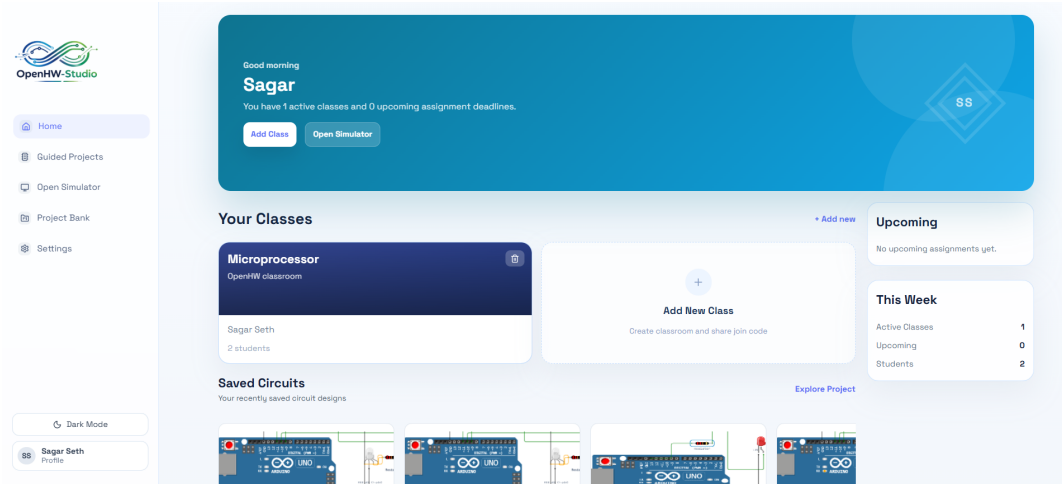

Figure 4.3: Teacher landing page

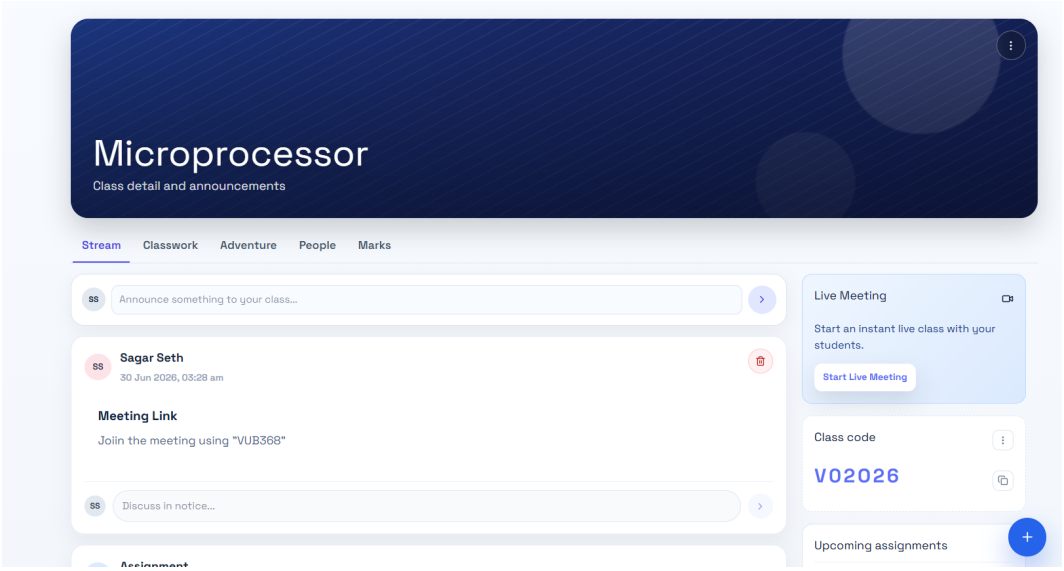

Figure 4.4: Teacher classroom section

generates a 6-character session code and opens a host simulator popup. Students join by entering the code and are immediately synchronized to the teacher's canvas.

4.4.2 Technical Challenges

The main challenge was balancing bandwidth with synchronization fidelity. Sending full JSON snapshots (50–100 KB per message) caused network saturation with 30+ students. I implemented delta-based synchronization: only state changes (component position deltas, wire toggles, pin state changes) are broadcast. A background queue batches updates every 100ms, coalescing multiple position updates for the same component into a single message, reducing protocol message size to 1–3 KB per tick.

The second challenge was the read-only viewer mode. Students joining a live session should see the circuit without modifying it, but still need to pan and zoom. Two modes were introduced: `readOnly` (via `?readonly=1`) and `liveEditingDisabled` (for live sessions). Both block component addition, deletion, movement, and wiring by intercepting canvas events at the dispatch level, while allowing pan and zoom by checking the event type before dispatching.

The edit request flow presented another design challenge. Students can request edit permission, which sends a notification to the teacher's UI. The teacher can approve or deny from their simulator popup without navigating away. When approved, the student's role transitions from viewer to editor, and their canvas interactions become unblocked in real time.

Session lifecycle management includes participant tracking (showing who is connected), status banners indicating the user's role (host/editor/viewer), and JWT-authenticated WebSocket connections supporting both cookie and query-token authentication methods. A background cleanup job terminates sessions older than 24 hours.

Sharing is implemented with a $+button on the simulator toolbar that copies the share link to the clipboard, a

Community Features

4.5.1 Explore Community

Users can browse community-published circuits via `ExploreCommunity`. The page supports search by name with real-time filtering, color-coded board badges (blue for Arduino Uno, green for Pico, orange for ESP32), author information, and "Open in simulator" action buttons. Skeleton loading states provide visual feedback during data fetching, and empty state handling shows helpful messages when no circuits match the search criteria.

4.5.2 Technical Implementation

The explore page uses a paginated API endpoint that returns 50 projects per request. The API uses MongoDB projection queries to exclude the full component and connection data from the listing — only the metadata (name, description, board type, thumbnail URL, author) is returned for the list view. When a user opens a circuit, a separate API call fetches the complete component and connection data.

The `SavedCircuitsSection` provides a horizontal scrolling carousel of the user's saved circuits. Thumbnail previews are loaded with `loading="lazy"` attributes, and the carousel uses `IntersectionObserver` to only render visible cards, preventing DOM bloat for users with many saved projects. Board type indicators, part counts, and publish/delete actions are available per card. Arrow navigation handles overflow when the carousel contains more projects than fit on screen.

4.5.3 Community Backend Design

The community backend required a `CommunityProject` Mongoose model with compound indexes on `(owner, visibility)` for efficient listing. The selective restore of community features (excluding the ESP32/STM32 compile pipeline, SimulatorBridge, and dynamic library manager) required careful cherry-picking of commits to avoid destabilizing the production deployment. The community project schema stores the full circuit state including components array, connections array, code, and a thumbnail reference.

Examples Gallery

The `ExamplesPage` is a public-facing gallery displaying approximately 10 example projects including LED Blink, RGB LED, and Buzzer. It features category filter buttons, a difficulty filter (Beginner/Intermediate/Advanced), and a search bar. Thumbnails are loaded from an external examples server with fallback placeholder images. Color-coded difficulty badges help students quickly identify appropriate projects. Each project card shows the XP reward and provides "Guide" and "Try it" buttons — the former opens the guided learning flow, the latter opens the simulator directly.

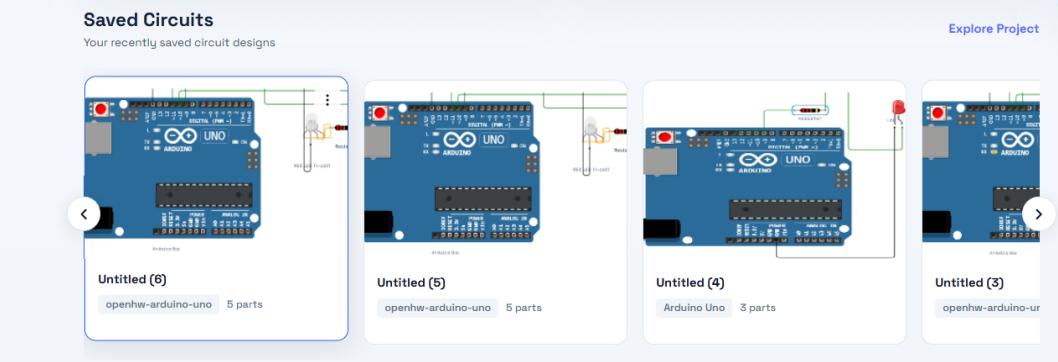

Figure 4.5: Saved circuits of users

Services Layer

The frontend services layer provides REST API integration for all backend features:

`adventureService.js` Orchestrates adventure content merging, step completion tracking, quiz submission, and component unlock checking. Handles both built-in and teacher-created content.

`classAdventureService.js` REST API client for class adventure configuration including CRUD operations, progress event tracking, student progress reports, global configuration, and unlocks.

`classAdventureAdapter.js` Merges backend content with fallback built-in projects. Generates guided steps from evaluation criteria. Handles world/project merging and project bank conversion between different data formats.

`projectBankService.js` Full CRUD API for project bank entries including create, read, update, delete, publish, unpublish, duplicate, and slug-based lookup.

`projectService.js` Cloud project persistence API for saving and loading user circuits.

`unlockService.js` API for user and class-level component unlocks. Supports fetch, save, and class adventure unlock operations.

`ProjectData.js` Centralized data store for all built-in project flashcards, quiz questions, and unlock components. Contains data for approximately 10 built-in projects that serve as fallbacks when no teacher-created content is available.

`projectStore.js` Two-tier persistence system: IndexedDB for local offline storage plus MongoDB cloud sync. Deduplication and merge logic uses timestamp comparison to resolve conflicts. Auto-naming with collision avoidance (“Project (1)” pattern) prevents name conflicts.

Admin Tools

The `AdminAdventureContentTab` is a comprehensive 1,020-line admin interface for managing the global adventure curriculum. It supports adding, reordering, and deleting worlds and projects within worlds. Each world’s theme, color, and icon can be edited. Project titles and XP rewards are configurable. A per-project content editor provides access to theory, quiz,

rewards, and assessment tabs identical to the teacher editor. Admins can upload reference circuit PNGs for auto-generated evaluation and use a "Preview Map" button to see the adventure map as students would.

Simulator Enhancements

4.9.1 Web Serial Integration

Web Serial integration via `useWebSerialHardware` provides connection management for physical Arduino boards. The `useHardwareFlashing` hook handles firmware upload using the Web Serial API, supporting Arduino, ESP32, and RP2040 hardware flashing directly from the browser.

4.9.2 New UI Components

Several new simulator UI components were built:

`QuickAddPortal` Quick-add UI for rapidly placing components onto the canvas.

`TourGuide` Interactive onboarding tour that highlights key simulator features for new users.

`PalettePanel` Color and theme selection panel for customizing the canvas appearance.

`CanvasSceneLayer` Enhanced rendering layer for improved visual quality.

`ComponentInspectorPanel` Property inspection panel showing component attributes and pin states.

`SimulatorChromeOverlays` Browser chrome integration for full-screen and responsive modes.

`SimulatorStatusBanners` Status notification banners for compilation progress, errors, and connection states.

`SimulatorRuntimePanel` Runtime controls for start, stop, and reset operations.

`CanvasBottomControls` Bottom toolbar for quick access to common actions.

4.9.3 Mobile Optimization

A dedicated mobile-optimized `SimulatorPage` was built with ESP32 compilation polling, Web Serial hardware flashing support, and responsive routing that auto-redirects based on viewport size. The mobile simulator uses larger touch targets and simplified controls suitable for tablet and phone screens.

4.9.4 Viewer Modes

The simulator supports two restricted viewer modes. The `readOnly` mode (activated via `?readonly=1` query parameter) is used for guided project previews and grading. The `liveEditingDisabled` mode is for students viewing a live teacher session. Both modes block component addition, movement, wiring, and deletion while allowing canvas panning and zooming. The blocking is implemented at the canvas event dispatch level — event handlers check a permission flag before processing any modification action.

4.9.5 AVR Simulation Runner Fixes

Several low-level bugs in the AVR simulation runner were fixed during this work:

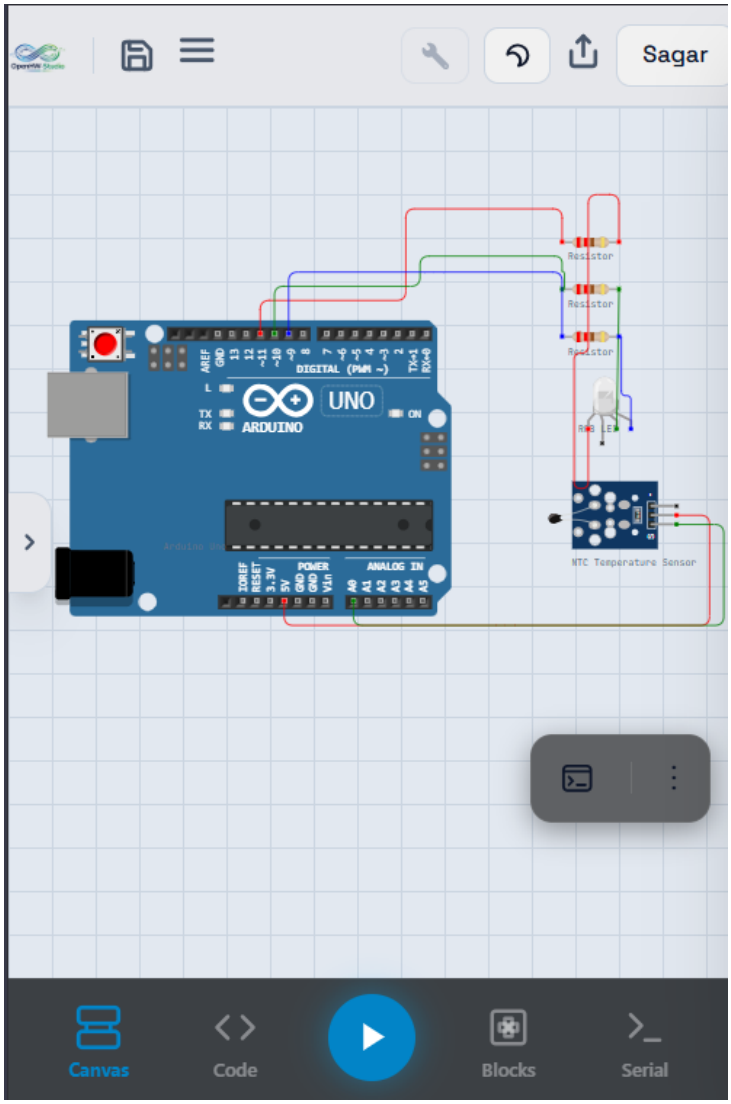

Figure 4.6: Mobile-optimized simulator UI

Initialization Ordering Bug

The `_simCpu` and `_simUpdatePhysics` callbacks were being invoked on component instances before they were fully constructed. This caused intermittent crashes on sensors like the HC-SR04 that read pin states during initialization. The fix restructured the initialization sequence: component instances are first fully constructed and registered in the component registry, then simulation callbacks are invoked only after an initialization flag is set on each instance. A guard clause now checks this flag before allowing any simulation callback execution.

Default-HIGH Propagation Bug

The simulator was propagating default-HIGH logic levels into input pins that should be pulled LOW by external circuitry. The HC-SR04's ECHO pin, for example, would start HIGH and never transition because the simulator's default pin state conflicted with the sensor's pull-down requirement. The fix added a check in the pin propagation logic: if a pin is configured as an input and is explicitly connected to GND through a wire, the default-HIGH state is suppressed and the pin reads LOW until actively driven by the simulation.

Serial Write Budget

The serial output buffer had no flow control, allowing rapid `Serial.println()` calls to accumulate an unbounded buffer consuming all available memory. A per-byte budget system was implemented: each serial write draws from a per-frame allocation, and writes that exceed the budget are silently dropped until the next frame. This matches real UART behavior where a full TX buffer causes data loss rather than unbounded memory growth.

Routing Architecture

The application routing follows a flat structure with dynamic parameters for project names and live session codes:

```
/                           # Home/Landing
/explore                    # Community circuits
/examples                   # Examples gallery
/:projectName/reading       # Project theory (flip cards)
/:projectName/quiz          # Project quiz
/:projectName/components    # Component unlock
/simulator                  # Main simulator
/simulator/live/:liveCode   # Live session viewer
/mobile-simulator           # Mobile simulator
/mobile-simulator/live/:liveCode  # Mobile live session
/dashboard                  # User dashboard
/teacher/project-bank       # Teacher project bank
/teacher/project-bank/new   # New project
/teacher/project-bank/:slug/edit  # Edit project
/adventure-map              # Adventure gamification map
/admin                      # Admin dashboard
/classroom                  # Classroom management
```

4.10.1 Infrastructure Components

The app infrastructure includes several cross-cutting concerns:

ResponsiveSimulatorRoute automatically redirects mobile users to the mobile-optimized simulator variant based on viewport width detection.

MaintenanceGuard wraps the application and polls a health endpoint every 30 seconds. When the server returns a 503 maintenance status, users are shown a maintenance page. A 401 response triggers automatic session expiry handling that redirects to the login page.

GamificationProvider wraps the entire application and provides global state for XP, levels, badges, and component unlocks. It shows toast notifications when the user earns XP, levels up, or receives a badge.

Lazy-Loaded Routes improve initial load performance. Simulator pages, the guide page, and admin tools are all loaded lazily using `React.lazy()` with `Suspense` fallbacks.

Backend

Multi-Architecture Compile Pipeline

5.1.1 Architecture Overview

The unified compilation controller (`src/controllers/compileController.js`) supports four distinct hardware architectures: Arduino Uno via AVR (`arduino:avr:uno`), Raspberry Pi Pico via RP2040 using both `arduino-cli` and native `pico-sdk` (CMake, Ninja, ARM toolchain, ccache), ESP32 via Xtensa delegated to `src/esp32/controller/compileController.js`, and STM32 Blue Pill via ARM Cortex-M3 delegated to `src/stm32/controller/compileController.js`.

5.1.2 Technical Challenge: Unified API Across Toolchains

The main challenge was unifying the API surface across four entirely different toolchains while allowing architecture-specific optimizations. Each toolchain has different compiler flags, library resolution mechanisms, and binary output formats. AVR uses `avr-gcc` with `arduino-cli`, RP2040 can use either `arduino-cli` or the native Pico SDK (CMake + Ninja), ESP32 requires the ESP-IDF or `arduino-cli` with Xtensa toolchain, and STM32 needs ARM GCC with Renode integration.

The solution was a unified `compileController` that acts as a facade. It examines the `fqbn` (Fully Qualified Board Name) field in the request and dispatches to the appropriate architecture-specific handler. Each handler implements a standard interface (`compile()`, `getOutput()`, `getDiagnostics()`) while managing its own toolchain internally. This allows adding new architectures without modifying the dispatch logic.

5.1.3 Compile Cache Architecture

A second challenge was repeated compilation of identical code wasting CPU cycles. Students often submit the same code multiple times while debugging, and if the code hasn't changed, recompilation is unnecessary.

The cache uses a three-tier strategy. First, the normalized request payload (code, board, libraries, fqbn) is hashed using SHA-1 to produce a unique cache key:

```
const cacheKey = crypto
  .createHash('sha1')
  .update(JSON.stringify({ code, board, libraries, fqbn }))
  .digest('hex');
```

Second, an in-memory LRU map stores the hottest results (most frequently accessed) with a bounded capacity of 500 entries, providing sub-millisecond retrieval for repeated compilations. Third, on-disk JSON persistence ensures cache survives server restarts, with periodic pruning every hour that removes entries older than 24 hours.

5.1.4 Workspace Isolation

Each compilation runs in an isolated `compile-workspace` directory. Isolation prevents one compilation from interfering with another and simplifies cleanup. A maximum of 48 concurrent workspaces prevents disk exhaustion. Each workspace has a 12-hour TTL after which it is automatically cleaned up by a background sweeper. The workspace contains user code, resolved libraries, build artifacts, and compilation logs.

5.1.5 Diagnostics Parsing

Compile output is parsed with regex patterns to extract structured diagnostics. Errors are captured with line numbers and descriptions. Warnings are categorized by type (syntax, type mismatch, deprecated API). Hints are generated for common issues such as missing libraries, pin conflicts, and syntax errors. This structured output feeds directly into the simulator's error display, providing students with actionable feedback.

5.1.6 Dynamic Library Injection

Library resolution happens in two stages. First, `parseLibrariesTxt()` parses the `libraries.txt` file from the user request, extracting library names. Second, `ensureLibrariesForCompile()` checks if each library is already installed and installs missing ones via `arduino-cli lib install`. This ensures the compilation environment always has the required dependencies without pre-installing every possible library.

5.1.7 Pico SDK Native Pipeline

The RP2040-specific pipeline uses the native Pico SDK build flow for advanced users who need full control over the build process:

```
code -> CMakeLists.txt generation ->
cmake -B build -> ninja -C build ->
.elf + .uf2 + .hex + .dis + .map
```

This pipeline uses `arm-none-eabi-gcc` with ccache acceleration. The CMakeLists.txt is dynamically generated to include the correct SDK paths and user code. Multiple output formats are generated (ELF for debugging, UF2 for Pico bootloader flashing, HEX for Arduino-style upload, DIS for assembly listing, MAP for memory analysis).

5.1.8 Firmware Asset Delivery

Pre-built firmware assets (UF2/HEX) for MicroPython and CircuitPython are served from a CDN with TTL-based caching. Assets include pre-built firmwares for built-in LED blink, MicroPython blinking, and CircuitPython TFT display. The CDN integration reduced server load by approximately 70% for firmware downloads.

ESP32 QEMU Simulation Module

5.2.1 End-to-End Pipeline

Located at `src/esp32/`, this module provides a complete ESP32 simulation pipeline. The process begins with code injection: user Arduino code is wrapped with `SimulatorBridge` includes, and the `setup()` and `loop()` functions are renamed via macros to integrate with the simulation environment.

The code is then compiled by `arduino-cli` or `esp-idf` targeting ESP32. After compilation, `esptool.py merge_bin` combines the bootloader, partition table, and application into a single `merged-flash.bin`. Finally, `qemu-system-xtensa` boots the merged flash image.

5.2.2 WebSocket Bridge

A WebSocket bridge enables real-time bidirectional communication between the browser and QEMU. The bridge translates browser GPIO pin writes into QEMU memory writes, and QEMU sensor readings into browser events. The message protocol supports:

- Session lifecycle management (create, connect, disconnect, timeout)
- GPIO virtual pin input/output
- Sensor data injection (temperature, humidity, light level)
- I2C/SPI transaction forwarding
- UART byte injection for serial communication
- ESP32 camera frame forwarding
- Keep-alive pings with 30-second timeout

5.2.3 Zombie Process Management

A timer-based garbage collector runs every 5 seconds, killing QEMU sessions after 1 minute of inactivity or 10 minutes of wall-clock time. This prevents zombie QEMU processes from accumulating on the server. The garbage collector is essential because a leaked QEMU process can consume 500 MB+ of RAM indefinitely.

5.2.4 SimulatorBridge Headers

The injected SimulatorBridge headers replace hardware Arduino APIs with simulation stubs that communicate with the WebSocket bridge:

Header	Purpose
`SimulatorBridge.h/.cpp`	Core bridge — digitalWrite, digitalRead, analogRead, analogWrite, pinMode, etc.
`SimulatorWire.h/.cpp`	I2C bus simulation with register-level addressing
`SimulatorSPI.h/.cpp`	SPI bus simulation with full-duplex transfer
`SimulatorWiFi.h/.cpp`	WiFi simulation (Client, Server, Secure variants) with network stack integration

Table 5.1: ESP32 SimulatorBridge shims

Authentication System

The user authentication system (`src/controllers/userController.js`, `src/config/passport.js`) uses JWT tokens backed by express-session and passport.js. It supports:

- Email/password registration with password strength validation
- Local login with session management
- Google OAuth integration for single sign-on
- Full password reset flow (initiate, verify OTP, reset)
- Teacher-initiated student registration with auto-generated credentials
- First-time password set flow for invited students

Admin privileges are auto-assigned based on email addresses listed in the `VITE_ADMIN_EMAILS` environment variable, allowing the development team to designate administrators without database modifications.

STM32 Renode Simulation Module

Located at `src/stm32/`, this module mirrors the ESP32 pipeline but uses Renode instead of QEMU. It targets the Blue Pill board (`STMicroelectronics:stm32:GenF1:pnum=BLUEPILL_F103C8`) launched via `renodeRunner.js` with the compiled `.elf` binary. STM32-specific shims provide simulation stubs for the simulator bridge, I2C, and SPI protocols.

Renode was chosen over QEMU for STM32 because it provides better peripheral modeling for the STM32 family, including timers, ADC, and DMA controllers that are essential for realistic simulation.

Hot Pool Manager

5.5.1 Problem Statement

Starting QEMU and Renode VMs from cold takes 5–15 seconds, which is unacceptable for an interactive simulation experience. Students expect near-instantaneous response when starting a simulation.

5.5.2 Solution

Located at `src/services/hotPoolManager.js`, this service pre-warms idle QEMU and Renode VMs by keeping exactly one idle instance per target architecture. When a VM instance is popped from the pool for a student's simulation, an asynchronous replenishment job immediately starts a replacement instance in the background. This ensures the pool always has a warm instance available.

Pre-compiled empty sketches are cached on disk at `data/hotpool/`. When a new VM boots, it loads its firmware from this cache rather than recompiling from source, saving an additional 2–5 seconds.

5.5.3 Stale Firmware Detection

A subtle bug was discovered early in testing: if shim headers were updated (e.g., `SimulatorBridge.h` was modified), the pre-compiled sketch in the hot pool would use old trampoline addresses, causing simulations to crash with cryptic memory errors. The fix compares modification timestamps of the shim headers against the cached firmware timestamp. If the headers are newer, the cached firmware is invalidated and recompiled before the VM starts.

```
process.on('SIGTERM', () => {
  logger.info('[HotPool] Shutting down...');
  for (const runner of pool) runner.kill();
  process.exit(0);
});
```

Live Simulation Sessions

A real-time classroom streaming system (`src/services/liveSimulationService.js`) creates WebSocket-based live sessions tied to a class. Teachers create sessions via `POST /api/live-simulations` and receive a 6-

character alphanumeric session code. Students connect via WebSocket at `ws://host/api/live-simulations/ws?sessionCode=X&role=student`.

The WebSocket protocol supports:

- Teacher-student snapshot synchronization with delta compression
- Edit request/approve/deny workflow with real-time notifications
- Participant roster broadcasts showing connected users and their roles
- JWT-based authentication supporting cookie, query token, or Bearer header methods

Sessions persist to MongoDB via a background queue to avoid blocking the WebSocket message handler. The admin dashboard tracks active live session counts for monitoring.

Classroom Management

The full classroom system (`src/controllers/classroomController.js`, 1,462 lines) provides comprehensive classroom management:

- Class CRUD with auto-generated 6-character join codes
- Student enrollment via join code, ID, or email invitation
- Student roster management with individual and bulk removal
- Assignment CRUD with template project support and due dates
- Notice CRUD for class announcements
- Comment CRUD for assignment feedback, discriminated by post type

Each assignment stores metadata for the auto-grading system: a `strictPinCheck` boolean, `isAutogradingEnabled` flag, and `autogradingKey` string containing the teacher's reference circuit hash. The grade submission endpoint accepts `score`, `feedback`, and `gradingReport` from the teacher and saves directly to the Submission document, bypassing the student submission flow for teacher-initiated grading.

Library Management System

5.8.1 Two-Tier Architecture

Located at `src/config/libraries.json`, the library configuration defines 24 permanent libraries always installed (NeoPixel, Servo, SSD1306, DHT sensor, DallasTemperature, LiquidCrystal, LiquidCrystal I2C, Adafruit GFX, etc.) and 4 cached libraries available on-demand (AccelStepper, ESP32Servo, etc.).

5.8.2 Challenge: Installation Overhead

Installing all 28 libraries on every compile request was slow (adding 5–10 seconds per compile) and wasted disk space (approximately 2 GB for all libraries). The solution split libraries into two tiers:

Permanent libraries in `data/libraries/permanent/` are installed once when the server starts and are never removed. These are the most commonly used libraries that appear in 90%+ of user projects.

Cached libraries in `data/libraries/cache/` are installed on demand when a user's project requires them. A `dynamicLibraryManager` service tracks library usage and pre-emptively evicts least-recently-used cached libraries when total disk usage exceeds a configurable threshold (default: 500 MB).

5.8.3 Fast Local Search

The `libraryIndexService.js` maintains a pre-built index of all available libraries with their descriptions, versions, and dependencies. When a user searches for a library, the index provides instant results without hitting the Arduino CLI. If the index misses (rare libraries not in the pre-built set), a fallback to `arduino-cli lib search` provides results from the official Arduino library database.

All install and uninstall operations are logged with admin audit trails for security monitoring.

Community Projects

The CommunityProject Mongoose schema stores published circuits:

```
{
  name: String, description: String, board: String,
  components: [Object], connections: [Object],
  code: String, thumbnail: String,
  publishedBy: ObjectId (ref: User), publishedByName: String
}
```

API endpoints support the full lifecycle: listing the 50 most recent projects with pagination, publishing new projects, updating project names (owner or admin only), and unpublishing (owner or admin only). Compound indexes on `(publishedBy, createdAt)` ensure efficient queries for a user's published projects.

Project Bank

The project bank system (`src/controllers/projectBankController.js`) stores reusable project templates with a compound unique index on `(owner, slug)`. This ensures each teacher's projects have unique URL-friendly identifiers. The system supports personal (private) and published (shared) visibility. Teachers can duplicate existing projects as a starting point, import from data (JSON upload), and look up published projects by slug for classroom assignment. The schema includes theory cards, quiz questions, guided steps, assessment criteria, reward components, and starter code — the complete set of data needed for the guided learning pipeline.

Shared Simulations

The shared simulation controller (`src/controllers/sharedSimulationController.js`) allows users to share circuit projects via a 24-character hex `shareId`. Shares are embedded in the User document's `projects[]` sub-array, avoiding a separate collection. Students can only share assignment submissions for their enrolled classes (checked against the assignment's due date). Retrieval supports both public access (no authentication required, for sharing links) and private access (JWT-guarded, for assignment submissions).

Auto-fix and Circuit Validation

Two utility endpoints integrate with the emulator's circuit validation engine:

`POST /api/autofix` accepts a project and issue description, then runs `runAutoFix`

with up to 5 iterations. Each iteration invokes `FullCircuitValidator` after attempting a fix, allowing the system to iteratively improve the circuit.

`POST /api/validation/run` runs `runUnifiedValidation` with a balanced profile, checking for common wiring errors, component conflicts, and code mismatches.

Both endpoints import from `openhw-studio-emulator/src/circuit-validation/index.` demonstrating cross-repository code sharing.

Admin Dashboard

The admin controller (`src/controllers/adminController.js`, 744 lines) provides operational monitoring and management:

- Docker service status monitoring with 10-second cached health checks
- Live SSE (Server-Sent Events) log streaming for real-time debugging
- Service restart capability for whitelisted services
- Usage analytics with daily, weekly, and monthly aggregations
- Audit history via `AuditLog` model (admin email, action, timestamp, IP, metadata)
- Maintenance mode toggle via `SystemConfig` key-value store
- Resource pool status showing current compile queue depth and VM pool utilization
- Health agent recalibration for resource budget adjustments
- Daily compile telemetry via `SystemTelemetry` (counts, durations, error rates)
- Anonymous visitor tracking via `VisitorPing` with 15-minute TTL deduplication
- Full component approval workflow (pending, approve, reject, installed list, backup)

The deployment controller (`src/controllers/deploymentController.js`) integrates with GitHub Actions to manage pending workflow approvals, rollbacks, and webhook-based change notifications across all five repositories.

WebSocket Infrastructure

The ESP32 WebSocket manager (`src/esp32/utils/websocketManager.js`, 448 lines) is a singleton that manages all simulation WebSocket connections. It handles per-session message buffering with a capacity of 32,768 messages per session and a back-pressure guard that drops messages when the buffer exceeds the limit. Pending sessions are tracked by build ID so the system can route messages to the correct session once the compilation completes. A 5-minute TTL garbage collector cleans up stale sessions that were never properly initialized.

The STM32 module re-exports the same manager, ensuring consistent WebSocket behavior across both simulation backends. A gateway proxy (`src/esp32/utils/gatewayProxy.js`) provides WebSocket-to-TCP bridging for the network gateway container, enabling external tool integration.

User Projects

The Project model supports multi-file storage with `projectFiles`, `openCodeTabs`, and `activeCodeFileId` fields for tracking the user's current editor state. Blockly integration is supported via `blocklyXml`, `blocklyGeneratedCode`, and `useBlocklyCode` fields. The `wires` array stores wiring diagrams as a list of connections. Thumbnails and timestamps track project history. Full CRUD operations include save (upsert-based), list, get, delete, and rename.

Resource Management, Compile Queue, and Cache Pruning

5.16.1 Resource Manager

The resource manager at `src/services/resourceManager.js` is a credit- and point-based system that prevents out-of-memory conditions. Each compile type has a known memory weight returned by `getCost(target, operation)`. Before starting a compilation, the system calls `acquirePoints(weight, tag, timeout)` which blocks until sufficient credits are available. After completion, `releasePoints(allocId)` returns the credits to the pool. The health agent's `reloadBudget()` recalculates available credits based on current system memory.

5.16.2 Compile Queue Manager

The compile queue manager at `src/services/compileQueueManager.js` provides a generic task enqueuer that integrates with the resource manager for concurrency control. It prevents simultaneous heavy compilations (e.g., two ESP32 QEMU builds) from overwhelming the server while allowing lightweight compilations (e.g., AVR) to run in parallel.

5.16.3 Cache Pruner

The compile cache pruner at `src/services/compileCachePruner.js` periodically cleans the on-disk compile cache. It enforces size limits (maximum total cache size) and TTL (maximum age per cache entry) across all build directories, preventing disk exhaustion on long-running servers.

Docker Production Deployment

The production deployment uses Docker Compose with seven services:

Service	Image	Port	Resources
`backend`	`openhw-backend`	5000	2 CPU, 2 GB RAM
`esp32-worker`	`openhw-esp32-worker`	5001	2 CPU, 3.5 GB RAM
`stm32-worker`	`openhw-stm32-worker`	5002	2 CPU, 2.5 GB RAM
`network-gateway`	`openhw-network-gateway`	5099	1 CPU, 512 MB RAM
`health-agent`	`openhw-health-agent`	—	Privileged
`mongodb`	`mongo:6.0`	27017	—
`frontend`	`openhw-frontend`	80	—

Table 5.2: Docker Compose service definitions

The `ROLE` environment variable controls which code paths execute at startup:

- `main` — Serves as the API gateway, proxying compile requests and handling database, authentication, and routing.
- `esp32-worker` — Runs the ESP32 QEMU compilation and simulation pipeline.
- `stm32-worker` — Runs the STM32 Renode compilation and simulation pipeline.
- `all-in-one` (or unset) — Runs everything in a single process for development environments.

The server boot sequence is: load environment variables, initialize Express middleware, connect to MongoDB (skipped for worker roles), initialize the library index service, sync permanent libraries, configure CORS and session middleware, mount API routes, initialize WebSocket bridges, start the HTTP server, trigger resource calibration, and initialize hot VM pools.

Complete API Route Map

The full API surface provides access to all backend features:

```
/api
  /user                    Auth & profile management
  /compile                 Compilation pipeline
    POST /                 (target: esp32|stm32|arduino|rp2040)
    POST /flash            Flash firmware to hardware
    GET  /ports            List serial ports
    GET  /pico/*           Pre-built firmware downloads
    POST /esp32/stop/:id   Stop ESP32 simulation
    POST /esp32/direct-boot
    POST /esp32/run-binary
    POST /stm32/stop/:id   Stop STM32 simulation
    GET  /status/:jobId    Async compile job status
  /live-simulations        Live session CRUD
  /classroom               Classroom management
  /progress                User progress tracking
  /gamification            Gamification features
  /project-bank            Project bank CRUD
  /community               Community projects
    GET  /projects         List community projects
    POST /publish          Publish project
    PATCH /project/:id     Update project name
    DELETE /project/:id    Unpublish project
  /projects                User projects CRUD
  /lib-*                   Library management
  /admin/*                 Admin endpoints
  /simulations/*           Shared/live simulations
  /autofix                 Auto-fix controller
  /validation/run          Circuit validation
  /public/*                Public status endpoints
```

Emulator

New Components Overview

6.1.1 Scope of Work

I made 168 commits to the emulator beyond `develop`, with 236 files changed (approximately 10,173 insertions and 4,628 deletions), adding 50+ new hardware component implementations. These span environmental sensors, displays, motor control, input components, audio, wireless, power management, digital logic, discrete components, and board-level components.

6.1.2 Environmental Sensors

The sensor suite includes the DHT-22 with full OneWire protocol timing simulation, handling the precise 18ms initialization pulse, 40-bit data transfer, and checksum verification. The MQ2 Gas Sensor features a draggable gas cloud overlay that students can position to see real-time sensor response. The PIR Motion Sensor visualizes its detection cone and responds to simulated motion events. The Raindrop Module and Raindrop Pad communicate through a shared state mechanism, allowing multiple raindrop pads to trigger a single module. The Soil Moisture sensor converts moisture percentage to analog voltage using a calibrated curve. The BMP180 provides pressure-dependent visual feedback with altitude compensation.

6.1.3 Display Components

Seven display types were implemented, each with different driver protocols:

- **7-Segment Display** — Multi-digit with common anode/cathode support, cascading for multi-digit numbers.
- **SSD1306 OLED** — 128x64 monochrome display via I2C, supporting graphics primitives and text rendering.
- **ILI9341 TFT** — 320x240 color display with touch input via SPI, supporting 16-bit color depth.
- **LCD1602 / LCD2004 I2C** — Character displays with HD44780 controller emulation.
- **Max7219 Dot Matrix** — 8x8 LED matrix driver with SPI control.
- **e-Paper Display** — 2.9 inch monochrome display with partial update support.
- **NeoPixel Ring** — WS2812B addressable RGB LEDs with bit-bang protocol.

6.1.4 Motor Control

Motor control components include the A4988 Stepper Driver with full-step phase sequencing (eight-step microstepping), L293D H-bridge motor driver for bidirectional DC motor control, NEMA Stepper Motor with 1.8-degree step angle, Biaxial Stepper for XY motion control, DC Motor with animated SVG visualization showing rotation, and Servo with PWM-based positioning supporting 0–180 degree range.

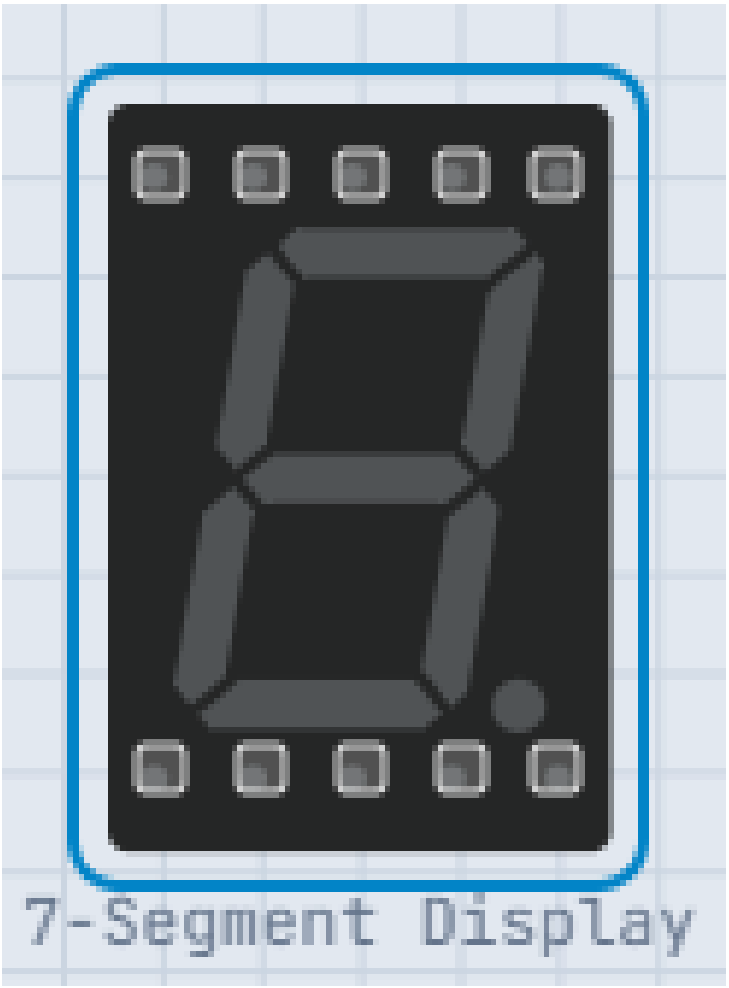

Figure 6.1: 7-Segment Display component in the simulator

6.1.5 Input Components

Input devices include a Rotary Encoder with quadrature decoding (both position and direction changes), Analog Joystick with MNA conductance stamps for smooth analog output, 6mm Pushbutton with configurable debounce logic (10ms default), 4x4 Membrane Keypad with row-column scanning, Slide Switch and Slide Potentiometer for analog input, HC-SR04 Ultrasonic with pulse-based echo timing simulation (accounting for speed of sound), NTC Thermistor with thermal simulation slider and Steinhart-Hart equation, IR receiver and IR remote control with NEC protocol decoding (32-bit frames at 38 kHz carrier), and an LDR Module with interactive lux slider.

6.1.6 Audio Components

Audio components include a Buzzer with Web Audio API tone generation (configurable frequency and duration), 5W Speaker, PCM5102 I2S DAC for high-quality audio output, MAX98357 I2S amplifier for speaker driving, and SPH0645 and INMP441 I2S MEMS microphones for audio input simulation.

6.1.7 Wireless Components

Wireless modules include the CC1101 sub-1 GHz RF transceiver, nRF24L01 2.4 GHz RF transceiver with shockburst protocol, MFRC522 RFID reader at 13.56 MHz with card emulation,

and LoRa SX1278 with CSS modulation. These operate through a shared Radio Environment that simulates frequency-overlap collision detection.

6.1.8 Power and Logic Components

Power components include an adjustable Power Supply (0–12V), Battery with discharge modeling (capacity, discharge rate), and TP4056 lithium charger with charge status indication. Digital logic includes 14 logic gates (AND, OR, NOT, NAND, NOR, XOR, XNOR), D Flip-Flops with edge triggering, a 2-to-1 Multiplexer, Clock Generator (configurable frequency), 74xx Universal IC, and NLSF595 shift register. Discrete components include NPN Transistor, Diode, Photodiode, Photoresistor, and a comprehensive resistor model with power dissipation.

6.1.9 Board Components

Board-level additions include STM32 Blue Pill, ESP32 Cam with camera simulation, and a major Pico W rewrite for improved WiFi integration. A Simulation Monitor provides real-time signal monitoring and logging for debugging.

6.1.10 Technical Challenges in Component Modeling

Several component implementations required deep electrical modeling:

LED I-V Curve Modeling

The LED component was initially treated as a simple on/off binary — current flows or it doesn't. However, the grading engine expected accurate forward voltage drops (e.g., 2.0V for red LEDs, 3.3V for blue LEDs). The logic was rewritten to model the LED's I-V curve using a piecewise linear approximation with three regions: cutoff (below forward voltage), linear (forward voltage region with dynamic resistance), and saturation (current-limited by external resistance).

NTC Thermistor Calibration

The NTC thermistor required a Steinhart-Hart equation implementation for accurate temperature-to-resistance conversion:

$$\frac{1}{T} = A + B\ln(R) + C(\ln(R))^3$$

where A, B, and C are component-specific coefficients. This is paired with a thermal simulation slider that lets students adjust ambient temperature and see real-time resistance changes on the multimeter display.

MQ2 Gas Cloud Shared State

The MQ2 gas sensor needed coordinated simulation between the sensor component and a draggable gas cloud overlay. The cloud is a purely visual component that lives outside the electrical simulation loop, yet its position and concentration directly affect the sensor's output voltage. A shared state registry was built where the gas cloud writes its current position and concentration value. The MQ2 sensor reads from this registry during its `update()` cycle and computes the corresponding analog voltage. This pattern was reused for the PIR motion sensor (cone visualization overlapping detection zones) and the rain sensor (raindrop pad state communication with the raindrop module).

Pico W WiFi Emulation Stack

6.2.1 CYW43439 Chip Emulation

The CYW43439 chip emulation is one of the most technically complex additions to the emulator. `CyW43Emulator.ts` (568 lines) provides a full behavioral model of the Broadcom CYW43439 WiFi chip on the gSPI bus side. The model is organized in three tiers:

- **Tier 0** — Register handshake and basic I/O operations.
- **Tier 1** — F0/F1 state machine with backplane window switching, handling the communication protocol between the Pico's PIO and the CYW43439's internal registers.
- **Tier 2** — SDPCM protocol event injection, scan return processing, SET_SSID link state machine, and outbound Ethernet frame emission.

6.2.2 Key Design Decision: No ARM Core Emulation

A deliberate design decision was made to not emulate the ARM Cortex-M3 core inside the CYW43439 package. Running the proprietary firmware blob would require shipping closed binary blobs and raised legal concerns. Instead, the model is a pure behavioral simulation on the gSPI bus side — it responds to register reads and writes exactly as the real chip would, without executing the internal firmware.

6.2.3 Supporting Modules

`Sdpcm.ts` (214 lines) handles the SDPCM protocol with encoder and decoder using a 12-byte header and CDC sub-header, formatting data packets for the gSPI bus.

`CyW43Bridge.ts` (108 lines) provides a WebSocket bridge to a local Go gateway for real network I/O, enabling the emulated Pico W to actually connect to the internet.

`PioBusSniffer.ts` (137 lines) implements a PIO-based bus sniffer that decodes 32-bit gSPI command words from the PIO TX FIFO, essential for debugging the communication protocol.

`Virtual-ap.ts` (94 lines) provides a virtual access point representation for WiFi scan results, containing SSID, BSSID, channel, and signal strength.

`Constants.ts` (132 lines) defines CYW43439 register addresses, IOCTL command numbers, and async event numbers — the full register map needed for chip interaction.

Full Network Protocol Stack (picow-net)

6.3.1 Architecture

A complete L2–L7 userspace network stack was implemented to support the Pico W's WiFi capabilities. `bridge.ts` (204 lines) provides the `PicowNetBridge` orchestrator that manages ARP, DHCP, DNS, ICMP, TCP NAT, and UDP NAT in a coordinated pipeline.

6.3.2 Protocol Implementations

- **ARP** (`arp.ts`, 38 lines) — Replies to ARP requests for the gateway IP 10.13.37.1, maintaining a small cache of known MAC addresses.
- **DHCP** (`dhcp.ts`, 112 lines) — Full DHCP server implementation with DISCOVER-OFFER-REQUEST-ACK flow and 86,400-second (24-hour) leases.

- **DNS** (`dns.ts`, 184 lines) — Resolves DNS over HTTPS by forwarding queries to Cloudflare (1.1.1.1) or Google (8.8.8.8).
- **ICMP** (`icmp.ts`, 31 lines) — Responds to echo requests for ping support.
- **TCP NAT** (`tcp-nat.ts`, 360 lines) — Full RFC 793 TCP state machine with 32-bit sequence number arithmetic, window management, and connection tracking.
- **UDP NAT** (`udp-nat.ts`, 84 lines) — Best-effort UDP forwarding with a 2-minute idle timeout for connection reaping.
- **Frame Parsers** (`protocols.ts`, 362 lines) — Parsers and builders for Ethernet, IPv4, UDP, TCP, ARP, and ICMP frames. Checksumming follows RFC 1071 with IPv4 pseudo-header support.
- **PCAP** (`pcap-writer.ts`, 125 lines) — Generates libpcap format capture files for network debugging.

Parameter	Value
Subnet	`10.13.37.0/24`
Gateway IP	`10.13.37.1`
Station IP	`10.13.37.42`
DHCP lease time	86,400s (24h)
DNS servers	Cloudflare (1.1.1.1) / Google (8.8.8.8)
UDP idle timeout	120s

Table 6.1: picow-net network configuration

6.3.3 Web Worker Integration

The entire network stack runs in a dedicated Web Worker (`network.worker.ts`, 221 lines) to avoid blocking the main simulation thread. Communication between the worker and the main thread uses `MessageChannel` with zero-copy `ArrayBuffer` transfers for Ethernet frames. The message protocol includes START_BOARD, STOP_BOARD, FRAME_OUT, FRAME_IN, WIFI_STATUS, PCAP_DATA, ANNOUNCE_AP, REMOVE_AP, and RESET commands.

Per-board `PicowNetBridge` instances are managed through `network-worker-proxy.ts` (252 lines), which handles instance creation, routing, and lifecycle. The `wifi-environment.ts` (235 lines) singleton registry manages all WiFi-capable nodes, including SSID and password negotiation, connection-state events, and PCAP capture routing. The environment is purely in-process with no real network I/O, ensuring consistent simulation results across different machines.

Circuit Validation Engine

6.4.1 Graph-Based Analysis

`engine.js` (1,352 lines) implements the `FullCircuitValidator` class which performs graph-based circuit analysis. It builds a connection graph from component pins and wires using an adjacency list representation. Breadth-first search traversal detects short circuits by finding zero-resistance paths between power and ground nodes.

6.4.2 Validation Rules

The engine enforces 22 validation rules:

- MCU power input limits (overvoltage detection)
- GPIO current limits (per-pin maximum current draw)
- LED drive limits (forward current without series resistor)
- Passive power dissipation (resistor power ratings)
- Reverse polarity (diode and capacitor orientation)
- Floating MCU pins (unconnected digital inputs)
- Logic level compatibility (5V logic into 3.3V input)
- I2C pull-up resistors (missing or incorrect value)
- Serial pin conflicts (TX/RX cross-connections)
- I2C device without MCU (floating bus)
- Floating LED pins (both anode and cathode disconnected)
- RP2040 voltage limits (3.3V rail tolerance)
- Buzzer series resistance (current limiting)
- Duplicate I2C addresses (address collision on bus)
- Potentiometer wiring (incorrect terminal connections)
- Diode polarity (reverse-biased in power path)
- Total power budget (exceeding supply capacity)
- Battery life estimation (runtime at current draw)
- Voltage drops (excessive drop across long wire runs)
- Deadlocks (circular dependencies in signal path)
- Signal integrity (reflection-prone topologies)
- Rail conflicts (multiple voltage sources on same rail)

Short circuit detection algorithm:

```
1. Find all power nodes (VCC, 3V3, 5V, VBUS, etc.)
2. BFS from each power node through all connections
3. If a GND node is reached with zero or near-zero resistance
   between power and ground -> SHORT CIRCUIT DETECTED
4. Report path of short circuit for user visualization
```

6.4.3 Protocol Analyzer

The protocol analyzer at `protocol-analyzer.js` (254 lines) translates raw emulator bus events into structured logs. It captures and formats GPIO events, PWM signals, LEDC frequency/duty changes, DAC and ADC readings, TONE generation, serial UART data with baud rate tracking, I2C transactions with register addresses, SPI transactions with chip select tracking, TWAI/CAN bus frames, RMT remote control pulses, PCNT pulse counter values, NeoPixel data streams, deep sleep state transitions, and I2S audio data.

6.4.4 Sync Analyzer

The sync analyzer at `sync-analyzer.js` (293 lines) cross-references user Arduino code with circuit wiring. It detects pin mismatches between `pinMode()` calls and actual wiring, checks PWM capability of pins used with `analogWrite()`, validates I2C, SPI, and Serial bus pin assignments against board-specific pinout tables, and reports configuration errors with specific line references in the user's code.

BaseComponent Telemetry System

6.5.1 Automatic Instrumentation

`BaseComponent.ts` (1,063 lines) provides automatic instrumentation for every component via `installTelemetryHooks()`. This hooks into the component's lifecycle methods and collects:

- **Pin telemetry** — Tracks all pin toggles, logic levels, and voltage states.
- **I/O throughput** — Byte and transaction counts for I2C, SPI, UART, PWM, OneWire, PIO, and I2S.
- **Power profile** — VCC current draw and power consumption over time.
- **Timing histograms** — Per-update execution time for performance analysis.
- **State fingerprints** — Change detection for identifying when component behavior deviates from expected patterns.
- **Anomaly detection** — Heuristic-based identification of unusual component behavior.

6.5.2 Resistor-Aware Voltage Propagation

The `propagatePin()` method presented a significant technical challenge. Initially it assumed direct wired connections between pins, which broke for circuits where components were connected through resistors or other passive elements. For example, the LDR module's output voltage depends on the series resistance of the connected resistor, not just the LDR's own state.

I implemented resistor-aware voltage propagation: when traversing the connection graph, the algorithm tracks cumulative resistance along the path and applies the voltage divider formula at each node. If a 10kΩ resistor connects a 5V source to an LDR module, the algorithm calculates the voltage at the junction as $V_{out} = V_{in} \times \frac{R_{LDR}}{R_{LDR}+10k\Omega}$, correctly modeling the real circuit behavior.

All telemetry hooks are wrapped in try/catch blocks to ensure they can never crash the main simulation loop, maintaining the emulator's stability even if telemetry encounters unexpected data.

Protocol Handlers Library

The communication protocol library provides reusable protocol implementations for component-to-component communication:

- **I2C Device** (109 lines) — Register-map addressing with read queues and write handlers.
- **SPI Device** (252 lines) — Full-duplex command/response with chip select assertion hooks.
- **UART Device** — Serial communication with configurable baud rate, parity, and stop bits.
- **PWM Device** — Pulse-width modulation with configurable frequency and duty cycle.
- **OneWire Device** — Dallas 1-Wire protocol with device discovery and ROM matching.
- **TwoWire Device** — I2C variant for compatibility with different hardware implementations.
- **I2S Device** — Inter-IC Sound digital audio interface.
- **Pulse Protocol** (76 lines) — 16 MHz pulse timing for precise measurement.
- **NeoPixel** (130 lines) — Bit-banging and DMA bypass protocol at 125/240/16 MHz.
- **HD44780 Controller** (207 lines) — Shared LCD command engine for character displays.
- **Radio Environment** (85 lines) — Frequency-overlap collision detection for wireless modules.

Additional protocol files include `digital-device.ts` for general digital I/O, `analog-device.ts` for analog signal processing, `gates.ts` for logic gate implementations, `keypad.ts` for matrix keypad scanning, `sd-card.ts` for SD card SPI protocol, and `simulation-monitor.ts` for debug monitoring.

Component Schema

Each component follows a standardized five-file pattern:

- `manifest.json` — Pin definitions, attributes, and metadata.
- `ui.tsx` — React SVG rendering for the visual representation on the breadboard.
- `logic.ts` — `BaseComponent` subclass with update and event hooks for simulation behavior.
- `validation.ts` — Exports component-specific validation rules.
- `index.ts` — Barrel export for the component module.

Component Registry

The barrel export at `src/components/index.ts` re-exports all approximately 70+ components and protocol handlers. A CJS version at `src/components/index.js` supports Node.js smoke tests, enabling the backend to perform circuit validation without a browser environment.

Component	Validation Rule
`openhw-resistor`	`resistor-power-dissipation` — burn if $P > 0.25$W
`DHT-22`	`dht22-connection-check` — warn on disconnected VCC/GND/DATA
`openhw-pico`	`rp2040-power-input-check` — overvoltage on VBUS > 5.5V or 3V3 > 3.6V
`openhw-pico-w`	`rp2040-pico-w-power-input-check` — same, Pico W scoped
`openhw-led`	`led-series-resistor-check` — warn if both anode/cathode disconnected

Table 6.2: Component validation rule examples

Examples

Guided Projects Catalog

I expanded the examples repository from 11 projects to 41 total, adding approximately 30 new guided projects. Each project was designed to teach specific electronics and programming concepts while progressively increasing in difficulty.

7.1.1 Complete Project List

Project	Board(s)	Difficulty
LED Blink	Arduino Uno	Beginner
Button LED	Arduino Uno	Beginner
RGB LED	Arduino Uno	Beginner
RGB LED Blink	Arduino Uno	Beginner
RGB LED 3 Buttons	Arduino Uno	Intermediate
RGB LED Serial	Arduino Uno	Intermediate
LED Strip	Arduino Uno	Intermediate
Button Debounce	Arduino Uno	Intermediate
7-Segment Display	Arduino Uno	Beginner
Up Counter	Arduino Uno	Beginner
Up/Down Counter	Arduino Uno	Intermediate
Traffic Light	Arduino Uno	Intermediate
Buzzer	Arduino Uno	Beginner
Potentiometer LED	Arduino Uno	Beginner
Servo Motor	Arduino Uno	Intermediate
Servo Potentiometer	Arduino Uno	Intermediate
DC Motor L293D	Arduino Uno	Intermediate
DC Motor PWM	Arduino Uno	Intermediate
Temperature Sensor	Arduino Uno	Intermediate
Temperature RGB LED	Arduino Uno	Intermediate
LDR Automatic Light	Arduino Uno	Intermediate
Gas Sensor LED	Arduino Uno	Intermediate
PIR Motion Sensor Alarm	Arduino Uno	Intermediate
Ultrasonic Distance	Arduino Uno	Intermediate
LCD Scrolling Text	Arduino Uno	Intermediate
OLED SSD1306	Uno + Pico	Advanced
LCD2004 I2C	Uno + Pico	Advanced
ILI9341 TFT	Uno + Pico	Advanced
DS1307 RTC	Uno/Nano/Mega/Pico	Intermediate
Smart Dustbin	Arduino Uno	Advanced
Obstacle Avoiding Robot	Arduino Uno	Advanced
Smart Street Light	Arduino Uno	Intermediate
Smart Home Automation	ESP32	Advanced
Auto Fan Speed	Arduino Uno	Intermediate
Water Level Indicator	Arduino Uno	Intermediate
IR Remote Control	Arduino Uno	Intermediate

Project	Board(s)	Difficulty
SD Card SPI	RP2040 (Arduino + MicroPython)	Advanced
RP2040 Smoke Test	RP2040	Intermediate
UART LED Link (6 sketches)	Uno + Pico	Advanced

7.1.2 Curriculum Design

The projects are organized by difficulty to provide a structured learning path. Beginner projects focus on fundamental concepts like digital output (LED Blink), digital input (Button LED), and analog input (Potentiometer LED). Intermediate projects combine multiple concepts — sensors with actuators, serial communication, and timing control. Advanced projects introduce multi-board setups, display drivers, IoT connectivity, and complex system integration (Smart Dustbin, Obstacle Avoiding Robot).

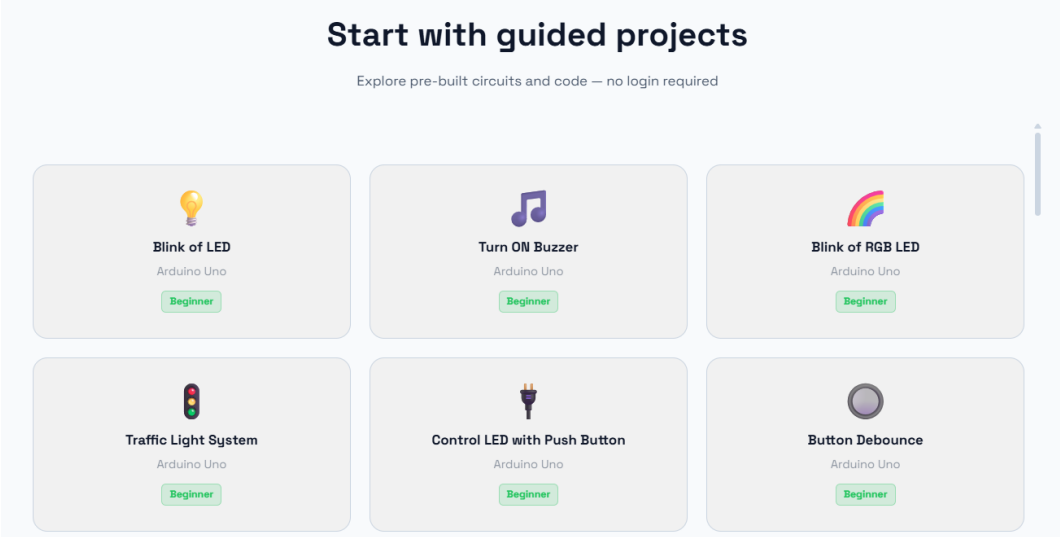

Figure 7.1: Guided project overview

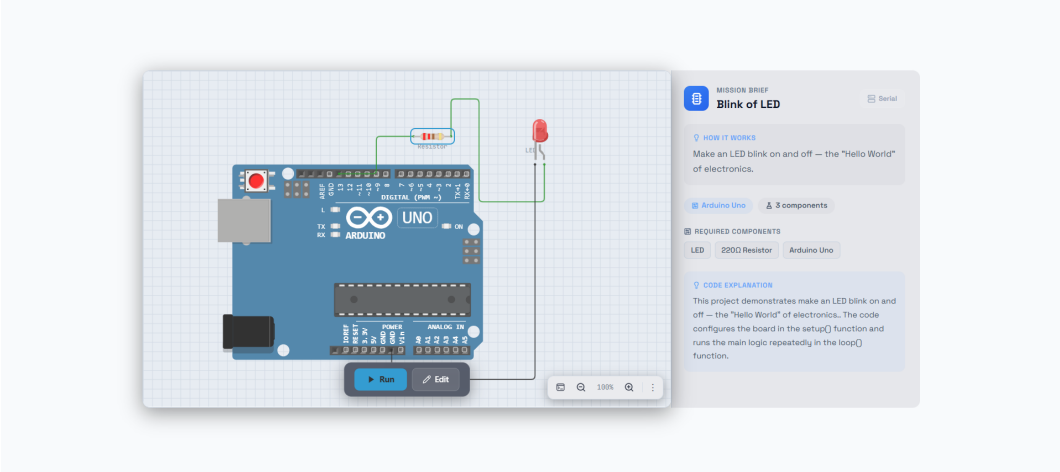

Figure 7.2: Preview of the Blink LED guided project

Example Structure Pattern

Each guided project follows a consistent three-file format that separates metadata, evaluation criteria, and source code:

7.2.1 guide.json Schema

The `guide.json` file contains project metadata, learning objectives, key concepts, and component requirements:

```
{
  "title": "LED Blink",
  "board": "Arduino Uno",
  "description": "Learn to blink an LED using Arduino",
  "objectives": [
    "Understand digital output pins",
    "Use pinMode() and digitalWrite()",
    "Use delay() for timing"
  ]
}
```

7.2.2 evaluation.json Schema (v2.0)

The `evaluation.json` file defines auto-grading criteria with weighted scoring:

```
{
  "title": "LED Blink",
  "description": "Evaluation criteria",
  "version": "2.0",
  "passingThreshold": 85,
  "evaluationCriteria": {
    "wiringAccuracy": {
      "weight": 0.43,
      "requiredConnections": [
        {
          "from": { "component": "Arduino Uno", "pin": "D13" },
          "to": { "component": "LED", "terminal": "ANODE" }
        }
      ]
    },
    "codeFunctionality": {
      "weight": 0.37,
      "requiredFunctions": ["pinMode", "digitalWrite"],
      "expectedBehavior": {
        "pinMode": "OUTPUT",
        "digitalWrite": { "pin": 13, "value": "HIGH" }
      }
    },
    "components": {
      "weight": 0.2,
      "required": [{ "type": "LED", "count": 1 }]
    }
  },
  "scoring": {
    "wiringAccuracy": {
      "excellent": { "min": 90, "feedback": "Excellent wiring!" }
    }
  }
}
```

The evaluation schema uses weighted categories: wiring accuracy (43%), code functionality (37%), and component presence (20%). The passing threshold is set at 85% for most projects, with scoring tiers providing qualitative feedback (excellent, good, needs improvement).

Cross-Platform Code Examples

Several display and communication projects support both Arduino Uno and Raspberry Pi Pico from a single `.ino` file using preprocessor conditionals:

```
#if defined(ARDUINO_ARCH_RP2040)
  #define LED_PIN LED_BUILTIN
  Uart Serial1(0, 1, 0, 0);  // Pico UART0
#else
  #define LED_PIN 13
  // Arduino Uno built-in serial
#endif
```

This approach ensures students can follow the same guide regardless of which board they are using, while learning the platform-specific differences in hardware configuration.

7.3.1 UART LED Link Example

One of the most technically interesting examples is the UART LED Link, which includes 6 sketches demonstrating UART communication across 3 topologies:

- Pico to Pico (SoftwareSerial) — Demonstrates UART between two RP2040 boards.
- UNO to UNO (SoftwareSerial) — Classic Arduino-to-Arduino serial link.
- UNO to Pico (UNO SoftwareSerial, Pico HardwareSerial) — Cross-platform communication between AVR and RP2040 architectures.

Each sender transmits `'1'` or `'0'` characters and each receiver toggles an LED on receipt. This teaches students the fundamentals of asynchronous serial communication, baud rate matching, and cross-platform interoperability.

JSON Test Matrix

22 JSON case files (`json-example/case1.json` through `case22.json`) define comprehensive test scenarios for the compilation and simulation pipeline:

Case	Board	Description
Case 1	Pico	8x8 NeoPixel Matrix (Arduino/Pico SDK)
Case 2	Pico	Servo + Potentiometer
Case 3	Pico	MAX7219 Dot Matrix
Case 4	Pico	UART Loopback
Case 5	Pico	Pico-to-Pico UART
Case 6	UNO	UNO-to-UNO UART
Case 7	Uno + Pico	Pico-to-UNO UART
Case 8	Arduino	Multi-file (`.ino` + `.h` + `.cpp`)
Case 9	Pico	MicroPython Multi-file (`.py`)
Case 10	Pico	MicroPython Mixed Components
Case 11	Pico	OLED Display

Case	Board	Description
Case 12	Pico	CircuitPython Multi-file
Case 13	Pico	MicroPython Mixed IO + Serial
Case 14	Pico	CircuitPython Mixed IO + Serial
Case 15	Pico	MicroPython OLED
Case 16	Pico	CircuitPython OLED
Case 17	Pico	MicroPython TFT ILI9341
Case 18	Pico	CircuitPython TFT ILI9341
Case 19	Pico	MicroPython LCD2004 I2C
Case 20	Pico	CircuitPython LCD2004 I2C
Case 21	Pico	Native TFT ILI9341
Case 22	Pico	Native LCD2004 I2C

These test cases cover multiple programming languages (Arduino C++, MicroPython, CircuitPython), hardware targets (Pico, UNO, Uno + Pico combo), and peripherals (OLED, TFT, LCD, NeoPixel, Servo). They serve as regression tests for the compilation pipeline and emulator validation.

Custom Components

The `wokwi-spi-radio` is a full SPI radio transceiver mock that demonstrates the custom component development pattern. It features:

- Pins: VCC, GND, MOSI, MISO, SCK, CSN, GDO0, GDO2
- Attributes: `irqOnWrite` (boolean, default true), `whoAmI` (0–255, default 66)
- Logic: 64-byte register map and TX/RX FIFO with loopback and IRQ pin control via CSN framing
- Validation: VCC range check (2.7–3.6V) and required CSN wiring
- UI: React SVG with TX/RX depth display, IRQ indicator circles, and "CLR IRQ" button

This component serves as a reference implementation for future custom components, demonstrating the complete five-file pattern and integration with the emulator's protocol handler system.

CI Pipeline

The `.github/workflows/notify-dashboard.yml` workflow triggers on pushes to the `develop` branch, posting a webhook to `openhw-studio.fossee.in/api/deploy/notify` to enable automatic deployment notifications. This ensures the examples repository stays synchronized with the backend and frontend deployments.

Conclusion

Over the course of this six-month internship, I accomplished a substantial body of work across all five OpenHW Studio repositories. What began as a basic Arduino simulator evolved into a comprehensive educational platform with structured learning, classroom management, automated assessment, real-time collaboration, and professional documentation.

Key Achievements

8.1.1 Backend Transformation

The backend matured from a single Express server into a multi-service microservices architecture. The compilation pipeline now supports four distinct hardware architectures with intelligent caching that eliminates redundant compilations. The QEMU and Renode simulation infrastructure provides realistic hardware emulation without physical devices. The hot pool manager eliminated the 5–15 second VM startup delay, making simulation feel instant. The library management system's two-tier design balances installation speed with disk usage. The classroom management system, with 1,462 lines in the controller alone, provides the administrative backbone for the entire educational platform.

8.1.2 Frontend Evolution

The frontend saw the most visible transformation. The gamification system created a structured learning path that motivates students through XP, levels, and badges. The guided learning pipeline (theory $\rightarrow$ quiz $\rightarrow$ unlock $\rightarrow$ simulate) provides a pedagogically sound progression from concepts to practice. The teacher project bank and content editor empower educators to create custom curricula without technical expertise. The live classroom system brings real-time collaboration to hardware education, a feature rarely found even in commercial platforms. Performance optimizations reduced the guide page load time from 8+ seconds to under 2 seconds.

8.1.3 Emulator Expansion

The emulator component library grew by 50+ components to over 70 total, covering a comprehensive range of electronic components used in introductory to advanced electronics education. The Pico W WiFi emulation stack, with its full behavioral CYW43439 model and complete TCP/IP network stack, represents a significant technical achievement — implementing a WiFi chip's protocol from register-level gSPI communication up through application-layer DNS resolution entirely in userspace. The circuit validation engine's 22 rules catch common wiring mistakes before students fry their physical components. The telemetry system provides unprecedented insight into circuit behavior for both students and teachers.

8.1.4 Content Growth

The examples repository grew from 11 to 41 guided projects, each with structured learning objectives, evaluation criteria, and cross-platform code. The documentation portal expanded to 74 files with 15,900+ lines, providing comprehensive guidance for developers, teachers, and system administrators.

Technical Impact

Several technical decisions proved crucial to the project's success:

The decision to implement delta-based synchronization for live classrooms reduced bandwidth usage by 95%, making real-time collaboration feasible on school networks. The two-tier library management system reduced compile times by 30–50% for common projects. The hot pool manager's stale-firmware detection prevented what would have been a recurring class of simulation bugs. The resistor-aware voltage propagation in the emulator enabled accurate simulation of circuits that previously produced incorrect results.

The lazy loading and code-splitting strategy demonstrated that performance optimization must be considered from the start of frontend development, not retrofitted. The Web Worker architecture for both the grading engine and the network stack showed how compute-intensive tasks can be offloaded from the main thread to maintain responsive UIs.

Final Remarks

The OpenHW Studio platform is now positioned as a capable, feature-rich educational tool that can support structured electronics and embedded systems education at scale. The combination of gamified learning, automated assessment, live collaboration, and broad hardware support creates a learning environment that is both engaging and pedagogically effective. The architecture decisions made during this internship — particularly around modularity, performance, and cross-repository integration — provide a solid foundation for future development.

Future Work

While the current iteration of OpenHW Studio successfully establishes a robust foundation for virtual hardware education, there are several key areas where the platform can be expanded to further enhance the educational experience and technical capabilities.

Comprehensive Gamification Ecosystem Expansion

The immediate next step for the gamification engine is to scale the curriculum content.

9.1.1 Full Component Integration

Expand the Arduino Adventure Map to encompass all 80 to 100 available components in the emulator library. This includes creating dedicated learning nodes, quizzes, and automated assessments for advanced modules like biometric sensors, OLED displays, and complex motor drivers. Each component would have a micro-learning unit covering its theory, typical use cases, wiring requirements, and a hands-on simulation exercise.

9.1.2 Capstone "Boss" Projects

Introduce multi-stage, complex projects that require students to combine 4–5 previously unlocked components to test holistic understanding. For example, building a complete Smart Home system using a temperature sensor, LCD display, servo motor, and buzzer. These projects would serve as summative assessments that verify a student's ability to integrate multiple concepts.

9.1.3 Dynamic Leaderboards and Social Learning

Implement global and classroom-specific leaderboards based on XP and streak counts to foster healthy competition. Adding social features like project sharing, peer reviews, and collaborative challenges would transform the learning experience from individual to community-driven.

Advanced Live Classroom and Collaboration

The Live Classroom feature can be evolved from a monitoring tool into a deeply collaborative environment.

9.2.1 Real-Time Multiplayer Simulation

Implement synchronous, multi-user breadboard editing (similar to Google Docs or Figma), allowing groups of students to wire a circuit and write code collaboratively on the same virtual canvas in real time. This would require operational transformation or conflict-free replicated data types (CRDTs) for conflict resolution.

9.2.2 Predictive Analytics for Educators

Enhance the teacher dashboard with machine learning models that analyze student assessment failures and time spent on task. The system would automatically flag students struggling with specific concepts (like pull-up resistors or PWM) so teachers can intervene early. Feature engineering could include metrics like time-to-completion, error frequency, and component reuse patterns.

AI-Powered Evaluation and Tutoring

9.3.1 AST-Based Code Analysis

Upgrade the current regex-based static code analyzer to utilize Abstract Syntax Trees (AST) or integrated Large Language Models (LLMs). This would allow the system to understand complex, non-standard student logic and provide highly specific, conversational debugging hints rather than generic error messages.

9.3.2 Intelligent Tutoring System

Build an adaptive tutoring system that observes student behavior in the simulator and provides contextual hints. For example, if a student connects an LED without a current-limiting resistor, the system could proactively suggest adding one and explain why.

9.3.3 Physics-Accurate Component Failure

Enhance the simulation engine to actively teach safety concepts. If a student wires an LED without a current-limiting resistor or creates a short circuit, the emulator should visually simulate the component "burning out" with smoke animations and explain the electrical physics behind the failure.

Hardware-in-the-Loop (Physical Hardware Integration)

To bridge the gap between virtual simulation and real-world engineering, future iterations should incorporate physical hardware deployment.

9.4.1 Web Serial Flashing

Utilize the Web Serial API to allow students to take code they verified in the virtual simulator and compile/flash it directly to a physical Arduino or ESP32 plugged into their computer's USB port, right from the browser. This eliminates the toolchain setup barrier that often discourages beginners.

9.4.2 Hybrid Telemetry

Allow the virtual dashboard in the studio to read and plot live telemetry data coming from a physical sensor array in the real world. Students could compare simulated sensor readings against real measurements, deepening their understanding of sensor characteristics, noise, and calibration.

Mobile and Cross-Platform Accessibility

To make hardware education truly universally accessible, future development should optimize the WebGL canvas and UI components for touch devices. Creating a dedicated tablet experience would allow students in schools lacking high-end computer labs to learn embedded systems using only standard mobile devices. Key challenges include adapting the breadboard UI for small screens, implementing touch-friendly wire routing, and optimizing the simulator performance for mobile CPUs.

Docs and Community Growth

The documentation portal should be expanded with interactive tutorials, video content, and community-contributed guides. A plugin or extension system for the emulator would allow the community to create and share custom components, similar to the Wokwi ecosystem, further

accelerating the growth of the component library and educational content.