Summer Internship Report

On

OpenHW-Studio (OSHW): Browser-Based Hardware Simulation & Educational Platform Development

Submitted by

Viraj Rahul Shah

B. Tech Computer Science Engineering

DES Pune University

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.

Viraj Shah

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.

Viraj Shah

Index

Chapter 1: Introduction

1.1 Project Overview

OpenHW Studio is an open-source, web-based hardware simulation platform developed under the **FOSSEE (Free and Open Source Software for Education)** project at **IIT Bombay**. It allows students, educators, hobbyists, and researchers to design, wire, and simulate Arduino-based electronic circuits entirely inside a browser — without needing physical hardware. By providing a virtual canvas of microcontrollers, sensors, displays, and actuators, OpenHW Studio lowers the entry barrier to embedded systems education and accelerates rapid Prototyping.

The platform offers a drag-and-drop component library, live wiring, real-time serial monitoring, and instant code execution. It is particularly well suited for STEM classrooms, online laboratories, and remote learning environments where access to physical Arduino kits may be limited. Components in OpenHW Studio are modeled with realistic pin-outs, electrical behavior, and configurable runtime attributes, enabling users to build and test projects ranging from simple LED blinks to advanced sensor-driven robotics systems.

By combining the flexibility of the Arduino ecosystem with a fully simulated hardware environment, OpenHW Studio fosters creativity, makes electronics prototyping more approachable, and supports India's wider push toward open, accessible, and high-quality engineering education.
The live deployment is accessible at https://openhw-studio.fossee.in/.

1.2 Scope of Contributions

During my summer internship, I proactively identified and resolved critical bottlenecks preventing the platform from transitioning from a local development environment into a production-ready, deployed web application.

My contributions spanned the entire stack. I led the **Docker containerization** of the monorepo, resolved severe cross-architecture compilation failures, and **established the secure deployment** networking on the FOSSEE VM. Furthermore, I conducted a comprehensive security audit of both: the backend, and

frontend API, redesigned core user interfaces, developed the platform's Blockly educational content and verified examples library, and expanded the hardware simulation ecosystem through new emulator components. Toward the end of my internship, I also architected a **robust backend testing framework** to safeguard the codebase against regressions as new contributors join the project.

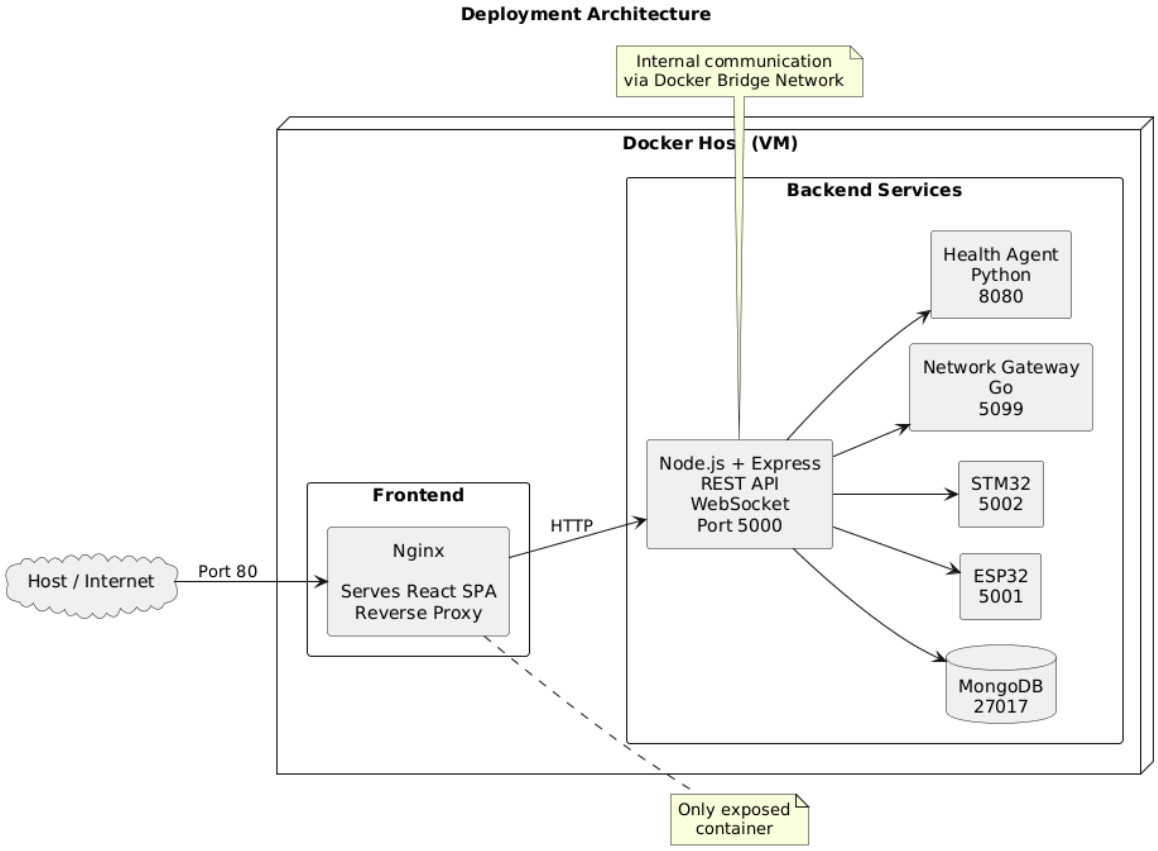
Chapter 2: Infrastructure & Deployment Architecture

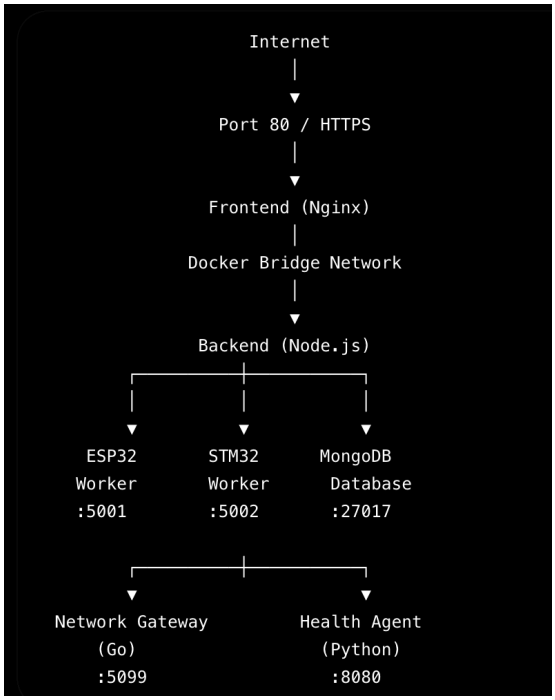
Transitioning OpenHW Studio from an asynchronous, local npm run dev application into an interconnected, live production environment required significant architectural restructuring.

2.1 Docker Containerization & Monorepo Architecture

Prior to my involvement, the application relied on local development environments, which suffered from hardcoded paths (e.g., Windows .exe paths for the arduino-cli). To lay the groundwork for all future updates and ensure cross-platform consistency, I containerized the entire ecosystem using Docker and Docker Compose.

I designed a container layout that maps one-to-one to the project's logical service boundaries:





Each service gets its own container with only the dependencies it needs. The frontend container uses Nginx to serve the static build and reverse-proxy API requests. All inter-service communication happens over a Docker bridge network — nothing is exposed to the host except port 80.

Managing a monorepo within Docker required precise configuration. I restructured the Dockerfiles to correctly set the build context, ensuring the frontend container had proper access to the openhw-studio-emulator and openhw-studio-examples directories via volume mounts and symlinks. This allowed the distinct parts of the application to build together without breaking container isolation. Furthermore, I optimized the backend Dockerfile to pre-install the arduino:avr core and essential libraries (Adafruit NeoPixel, Servo), eliminating massive network latency during the toolchain's first-run execution.

I wrote Dockerfiles for each service. The frontend Dockerfile uses a multi-stage build: -

- **Stage 1 (Build):** A Node.js image installs dependencies and runs npm run build, producing the optimized Vite output in dist/.
- **Stage 2 (Serve):** An Nginx Alpine image copies only the dist/ folder and a custom nginx.conf, resulting in a final image that's ~25MB instead of ~1GB.

The result of this effort is visible in the docker ps output — seven containers, all healthy, orchestrated from a single docker-compose up command:

```
admin@ubuntu24-internship:~$ docker ps
CONTAINER ID   IMAGE
 COMMAND                  CREATED          STATUS                   PORTS
                        NAMES
52bfcbfe9616   danish9060/openhw-frontend:16d5967911aa26a786c1d551211236aa9a89a8e8
 "/docker-entrypoint.…"   22 hours ago     Up 22 hours              0.0.0.0:80->80/tcp, [::]:80
->80/tcp                openhw-frontend
ce081c3ba526   danish9060/openhw-network-gateway:659719badf876634d09b56ed126f579f94e32dbe
 "./openhw-gw"            6 days ago       Up 6 days                0.0.0.0:5099->5099/tcp, [::
]:5099->5099/tcp        openhw-network-gateway
8c1276053604   danish9060/openhw-backend:659719badf876634d09b56ed126f579f94e32dbe
 "docker-entrypoint.s…"   6 days ago       Up 6 days (healthy)      5000/tcp
                        openhw-backend
9bfe313d5d1c   danish9060/openhw-stm32-worker:582cfec71a616523978d4c0ee44676b027a33645
 "docker-entrypoint.s…"   9 days ago       Up 9 days                5002/tcp
                        openhw-stm32-worker
1dd8ca7c6d67   danish9060/openhw-esp32-worker:582cfec71a616523978d4c0ee44676b027a33645
 "docker-entrypoint.s…"   9 days ago       Up 9 days                5001/tcp
                        openhw-esp32-worker
c1fabd82efbb   danish9060/openhw-health-agent:092096b8fcc311197aaa5f0c2e6219e128fb90a8
 "python health_agent…"   3 weeks ago      Up 2 weeks
                        openhw-health-agent
7e86ffeb1a9e   mongo:6.0
 "docker-entrypoint.s…"   2 months ago     Up 2 weeks (healthy)     0.0.0.0:27017->27017/tcp, [
::]:27017->27017/tcp    openhw-mongodb
admin@ubuntu24-internship:~$
```

Docker containers running in production (updated and built on top of by team).

2.2 Resolving Architecture Mismatches

Building an application on an Apple Silicon M4 chip and deploying it directly to a Linux server creates severe compatibility issues due to ARM64 versus AMD64 architecture differences.

During early deployment attempts, the avr-gcc compiler failed silently on the Linux VM. I diagnosed this as a libc compatibility issue: the initial Docker images were built on Alpine Linux (musl libc), which could not execute the Arduino toolchain binaries compiled for standard Linux. By migrating the Docker base image to Debian Slim (node:20-slim using glibc), the containers acted as a universal translator, ensuring the code compiled locally executed flawlessly on the target Linux machine without manual dependency wrangling.

2.3 Secure Access, Jump Hosts, and Nginx Reverse Proxy

Accessing the VM hosted on the FOSSEE Jupiter server required navigating strict security protocols. We utilized an AWS EC2 instance as a jump host. I configured

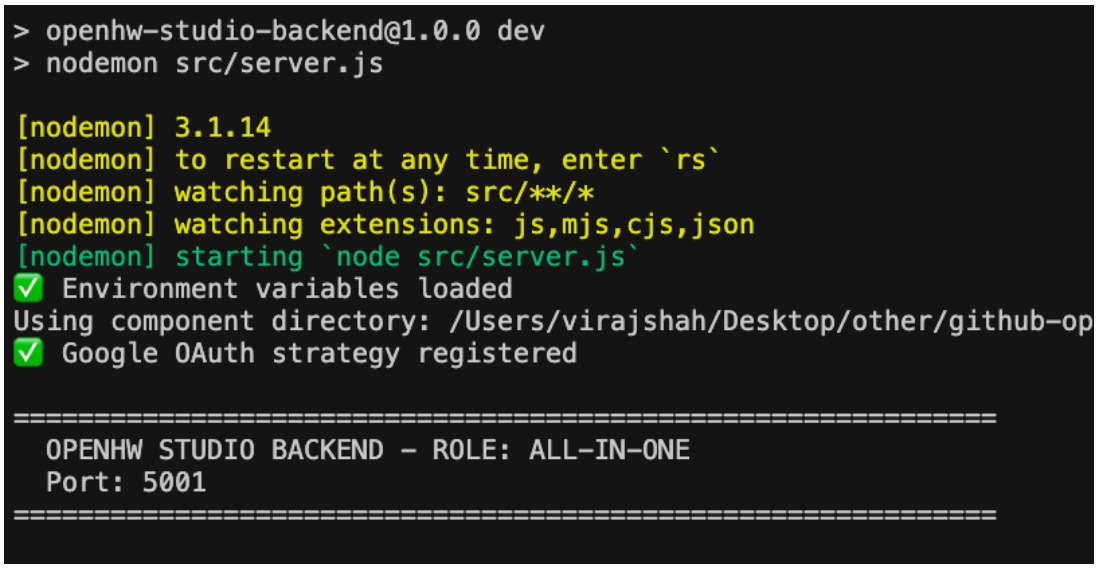
the SSH protocols to navigate this digital airlock, authenticating through the EC2 instance before gaining an escorted path into the internal Jupiter VM network.

Furthermore, raw port mapping is insufficient for a production-grade deployment. To make the application securely accessible via the official domain (openhw-studio.fossee.in), I implemented and configured an Nginx reverse proxy. This caught incoming external web traffic on ports 80 (HTTP) and 443 (HTTPS) and cleanly routed it to the correct internal Docker containers serving the frontend and APIs, keeping internal backend ports entirely hidden from the public internet.

The Nginx configuration handles two concerns:

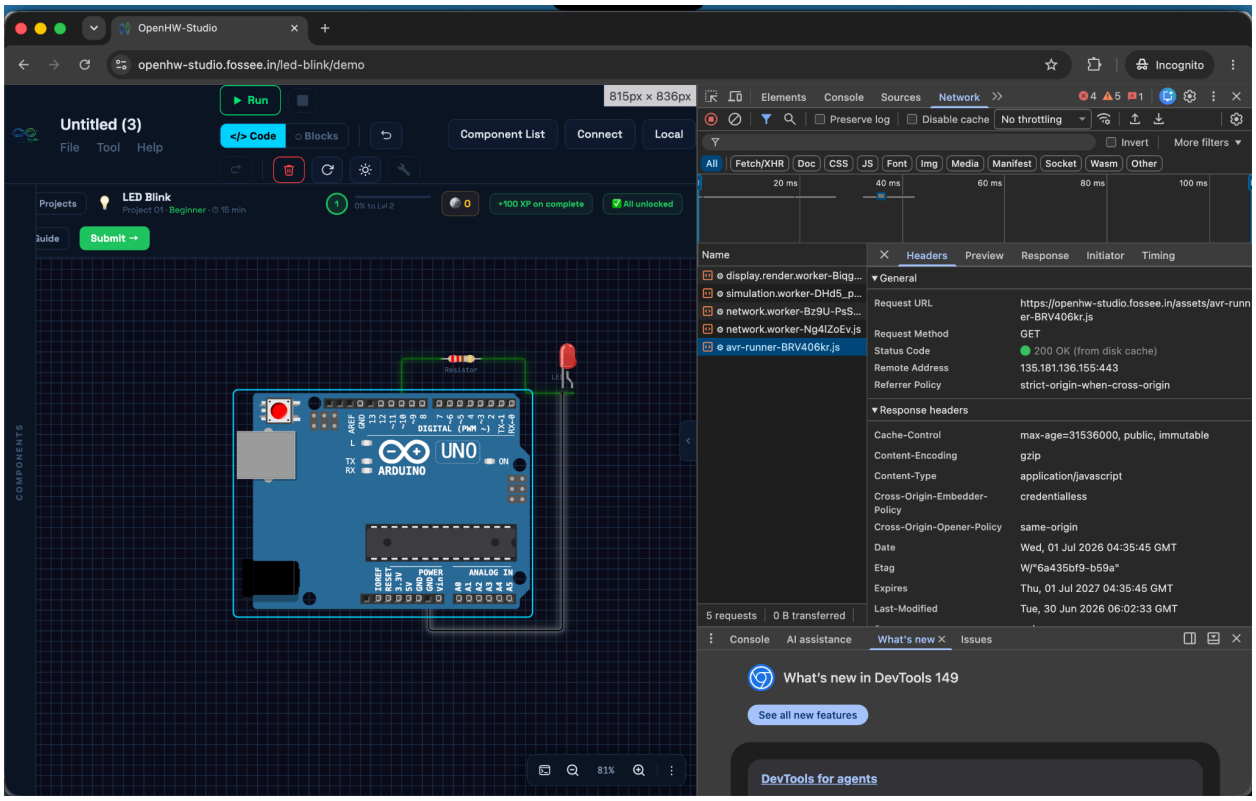
1. **Static file serving:** The Vite build output is served directly with proper try_files fallback so that client-side routing (React Router) works — any path that doesn't match a static file falls back to index.html, preventing 404s on direct URL access or page refreshes.
2. **API reverse proxy:** Requests to /api/*, /auth/*, and WebSocket upgrade requests (/api/live-simulations/ws) are proxied to the backend container. The config also handles WebSocket upgrade headers, which are required for the live simulation feature.

After deploying, I verified the full pipeline. The backend logs confirmed MongoDB connectivity, WebSocket bridge initialization for both ESP32 and STM32, and HotPool resource allocation responding to requests:



Backend startup logs showing successful initialization

The simulator loaded correctly in the browser, with the AVR runner JavaScript chunk returning 200 OK — proving Vite's code-split production build resolves correctly through Nginx:



Simulator running with AVR runner loaded successfully

2.4 CI/CD Pipeline Groundwork

To eliminate the friction of manual jump-host deployments, I architected the initial CI/CD pipeline groundwork using GitHub Actions. This automated workflow was designed to ensure that pushing code triggered cross-platform builds and handled remote server deployments, removing the need to manually execute Docker commands over SSH for every update.

Chapter 3: Backend Architecture & Security Hardening

With the application successfully containerized, my focus shifted to ensuring the backend APIs were secure, performant, and resilient against unauthorized access.

3.1 Comprehensive Security Audit & Data Leak Prevention

I conducted a thorough security audit of the Node.js backend, identifying and patching over 12 vulnerabilities. The platform, being developed by multiple contributors in an academic setting, had accumulated significant technical debt in the form of development-era artifacts that posed real security risks in production.

1. **Data Leak Prevention:**

I found and removed over 118 console.log statements across the backend codebase. These weren't just harmless debug prints — several logged MongoDB connection URIs (which contain passwords), user email addresses, full JWT payloads, and entire req.user objects into the server's standard output logs. In a production environment, this is noisy at best and a critical security liability at worst. If those logs are ever exposed — through a misconfigured log aggregator, an unauthorized access incident, or even a verbose error response — it's a direct credential and PII leak.

Specific critical fixes included:

- **JWT Payload Leakage:** authMiddleware.js and sharedSimulationController.js contained console.log statements dumping full JWT payloads to server logs. JWTs exist precisely so you don't need to log their contents — you verify the signature and trust the claims. I removed these statements entirely.
- **MongoDB URI Exposure:** Connection strings containing database credentials were being logged during startup. I removed these and ensured only connection status (success/failure) is logged.
- **Hardcoded Secrets:** A Google OAuth JSON secret file (client_secret_...json) was found committed directly to the repository root. This was flagged for removal and rotation.

2. **Timing Attack Prevention:**

The webhook endpoint used basic string comparison (===) to verify the incoming secret against the stored one. This is susceptible to timing attacks — an attacker can measure response times to incrementally guess the correct secret character by character, because === returns false as soon as it finds a mismatch (short-circuit evaluation). I replaced it with crypto.timingSafeEqual, which always compares the full string length regardless of where the mismatch occurs, eliminating the timing side channel.

3. **Potential Attack Surface Analysis:**

Beyond the specific vulnerabilities I patched, I assessed the broader attack surface:

- **DoS/DDoS Risk via Unauthenticated Endpoints:** The compilation routes were completely open (detailed in 3.2). A simple script could flood /api/compile/start with requests, exhausting server resources and denying service to legitimate users.
- **Resource Exhaustion via Simulation:** The WebSocket-based live simulation endpoints, if unauthenticated or unrate-limited, could be abused to monopolize HotPool simulation points, preventing other users from running simulations.
- **Information Disclosure via Error Responses:** Several error handlers returned raw stack traces or internal error messages to the client, which could reveal implementation details useful to attackers. These should be replaced with generic error responses in production.

3.2 Endpoint Hardening & Guest Compilation Optimization

The platform's compilation endpoints (/compile, /diagnostics, /flash) and several admin component endpoints had no authentication middleware. This meant anyone on the internet could submit compilation jobs to the backend, potentially exhausting server resources (a simple DoS vector) or accessing admin-only functionality.

I secured these routes by adding the existing protectRoute and requireAdmin middleware. However, I recognized that requiring a login to compile code would harm the platform's accessibility for beginner students — one of OpenHW Studio's core educational mandates.

To solve this, I documented and laid the groundwork for a protectRouteOrGuest middleware. This would allow anonymous users to access the web simulator safely while completely skipping the JWT verification and MongoDB database lookup steps for guest requests. This optimization would shave 20–50ms of database latency off every single compilation request made by guest users, while still applying rate limiting to prevent abuse.

3.3 Authentication & Google OAuth Integration

To streamline user onboarding, I worked on the initial implementation of the Google OAuth backend infrastructure. This involved:

- Configuring Passport.js strategies for Google OAuth 2.0
- Resolving redirect URI mismatches between local development (localhost:5000) and production (openhw-studio.fossee.in) environments
- Handling edge-case server crashes caused by bcrypt attempting to hash empty passwords for users authenticating via OAuth (OAuth users don't have local passwords, so the password field must be handled conditionally)
- Ensuring the JWT generation and cookie-setting logic worked correctly across both authentication paths

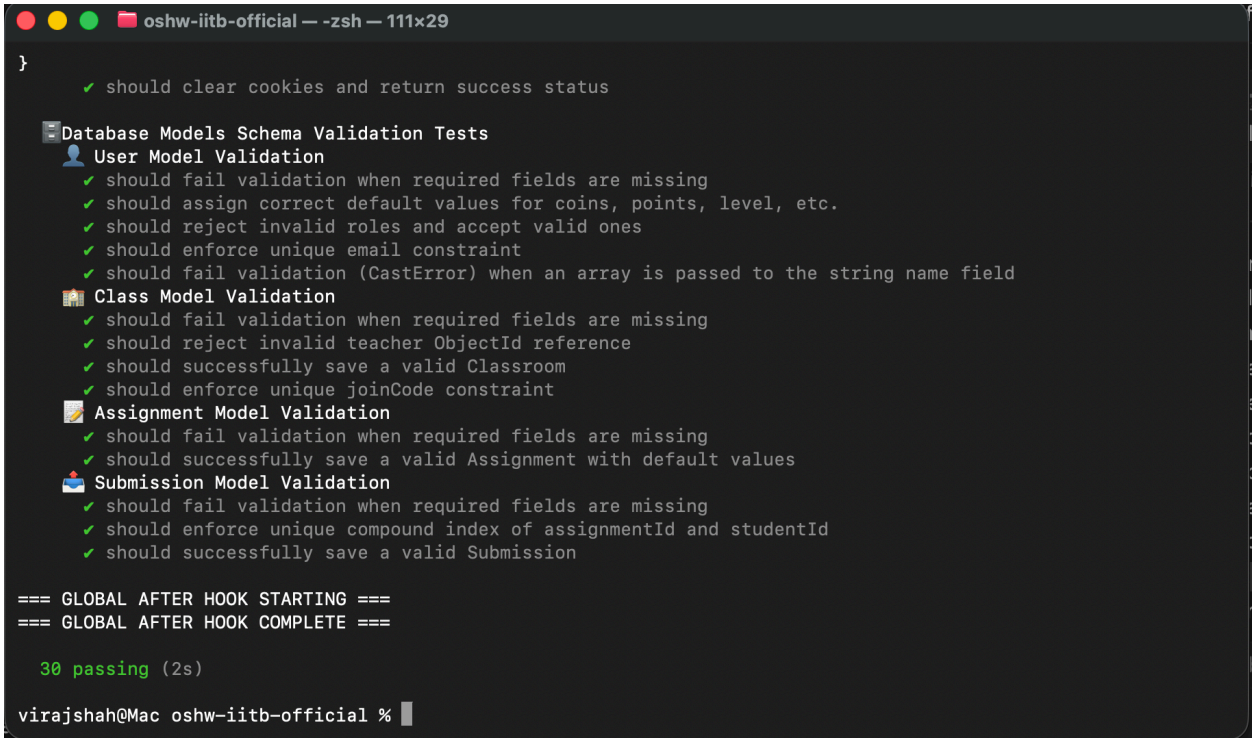
3.4 Backend Testing Infrastructure & CI/CD Integration

During a review meeting with Prof. Prabhu Ramchandran, a critical concern was raised: as OpenHW Studio grows and new contributors begin submitting code, the lack of automated testing meant a single poorly written Pull Request could silently break the entire backend. The system was fragile. To address this, I architected and implemented a comprehensive backend testing framework designed to act as an impenetrable guardrail against regressions.

I introduced Mocha, Chai, and Supertest as the core testing utilities, and implemented a mongodb-memory-server to allow tests to run against a temporary, in-memory MongoDB instance. This ensured that our test suite could execute in under 3 seconds without polluting the production database or requiring external container setup.

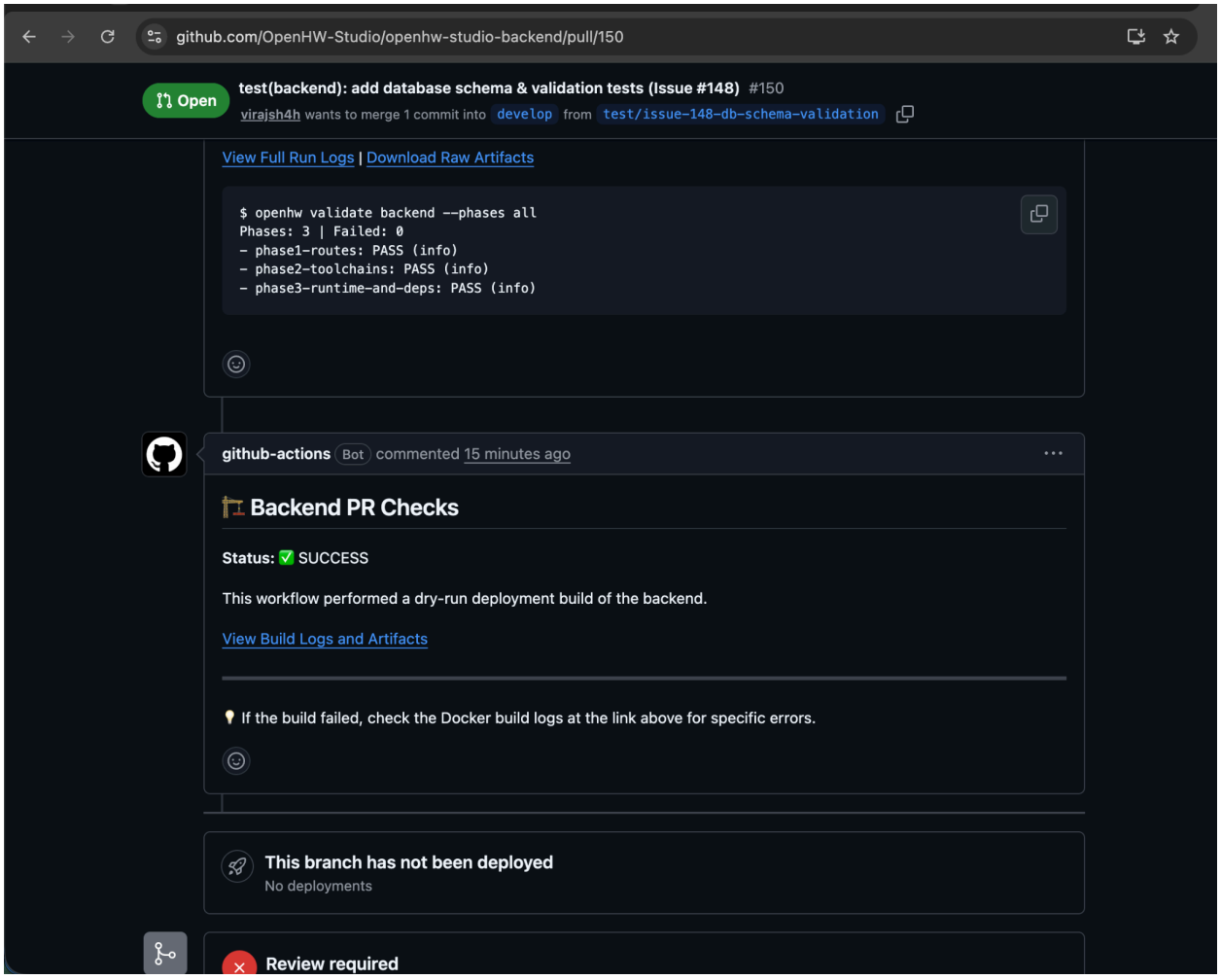
I developed two primary suites of tests:

1. **API Integration & RBAC Tests (Issue #147):** I wrote 16 tests verifying the core authentication flows (signup, signin, profile retrieval, logout). More importantly, I implemented Role-Based Access Control (RBAC) guardrails. I wrote explicit tests ensuring that unauthenticated requests return 401, student-level accounts attempting to access admin endpoints return 403, and admin accounts are successfully authorized. If a contributor accidentally removes the protectRoute or requireAdmin middleware from a route, these tests will fail instantly.
2. **Database Schema Validation Tests (Issue #148):** Bypassing the API layer, I wrote 14 deep tests targeting the raw Mongoose models (User, Class, Assignment, Submission). I verified required field validations, enum constraints (e.g., rejecting an invalid role like 'superadmin'), and database-level uniqueness constraints (E11000 duplicate key errors for emails and join codes). I also tested edge cases like MongoDB CastErrors for invalid ObjectId references and compound indexes to prevent students from submitting the same assignment twice.



Command npm test showing passing tests in ~2s.

Finally, I integrated this testing suite directly into the GitHub Actions CI/CD pipeline. I created a backend-tests.yml workflow that automatically spins up a Node.js environment on every Pull Request, runs the Mocha test suite, and physically blocks the PR from being merged if any test fails. This transformed the backend from a fragile system into an architecturally protected one.



Integration in Github Actions (CI / CD Workflow)

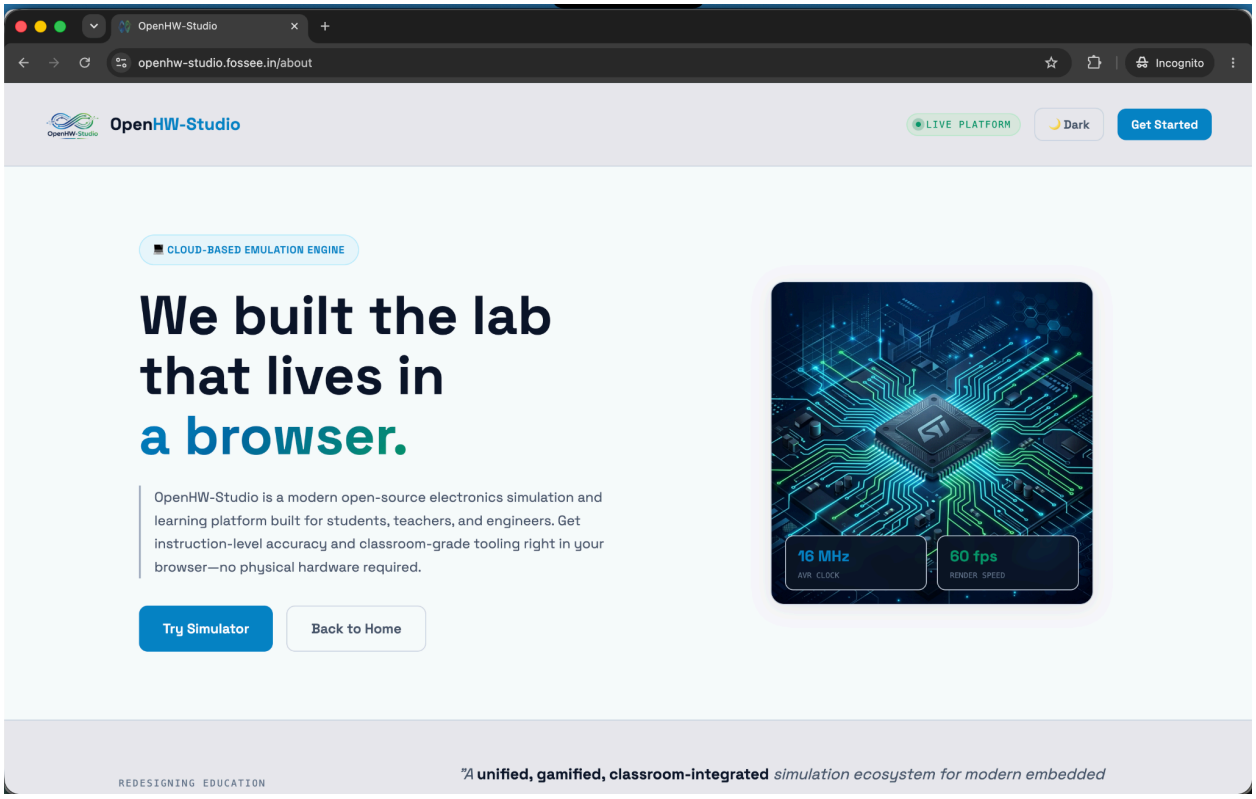
Chapter 4: Frontend Modernization & Hardware Accessibility

To complement the backend stability, I overhauled several user-facing interfaces to align with modern design principles and significantly improved accessibility for non-technical users.

4.1 UI/UX Overhaul & The “About Us” Redesign

I led a visual audit of the platform, identifying pages with broken layouts and empty states. Several pages on the platform had broken or incomplete designs — these weren’t minor polish issues but layouts that were visibly broken, with misaligned elements, missing styles, or placeholder content that made the platform feel unfinished.

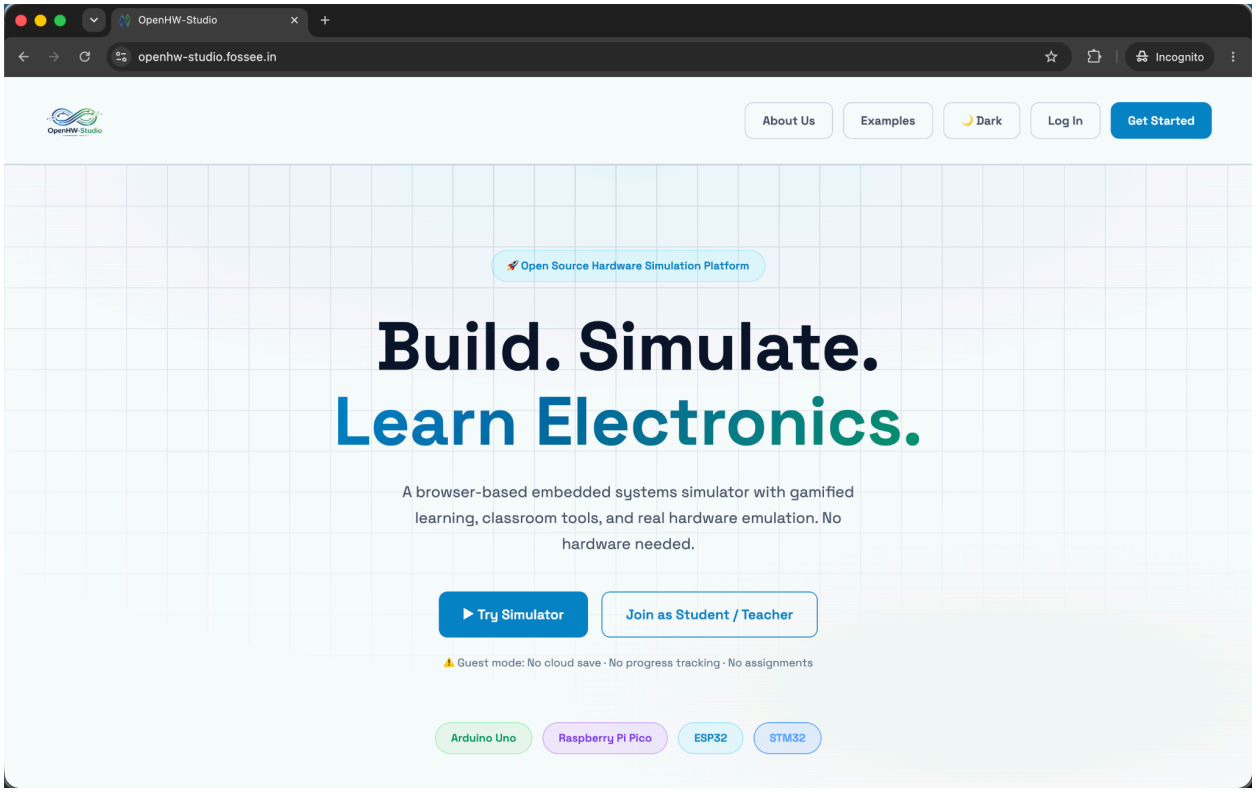
The most significant update was the complete redesign of the “About Us” page at https://openhw-studio.fossee.in/about. I deprecated a monolithic, inline-CSS legacy file and rebuilt it into a modern, responsive React SPA utilizing glassmorphic UI elements — frosted-glass card panels, layered gradients, and clean typography. The page now properly presents the team structure, roles, and the project’s mission.



Redesigned About Us page with a modern browser-based layout.

A technical detail worth noting: because the page uses highly custom styles that don't map cleanly to Tailwind's utility classes, I injected a self-contained `<style>` block on mount to bypass Tailwind's JIT scanning limitations. Tailwind only generates classes it finds in the source code at build time, so dynamically constructed class names or one-off design treatments need this approach.

The homepage had a broken layout "for so long" — elements were misaligned, the hero section wasn't rendering properly, and the overall visual hierarchy was off. I fixed the layout, resulting in the current state:

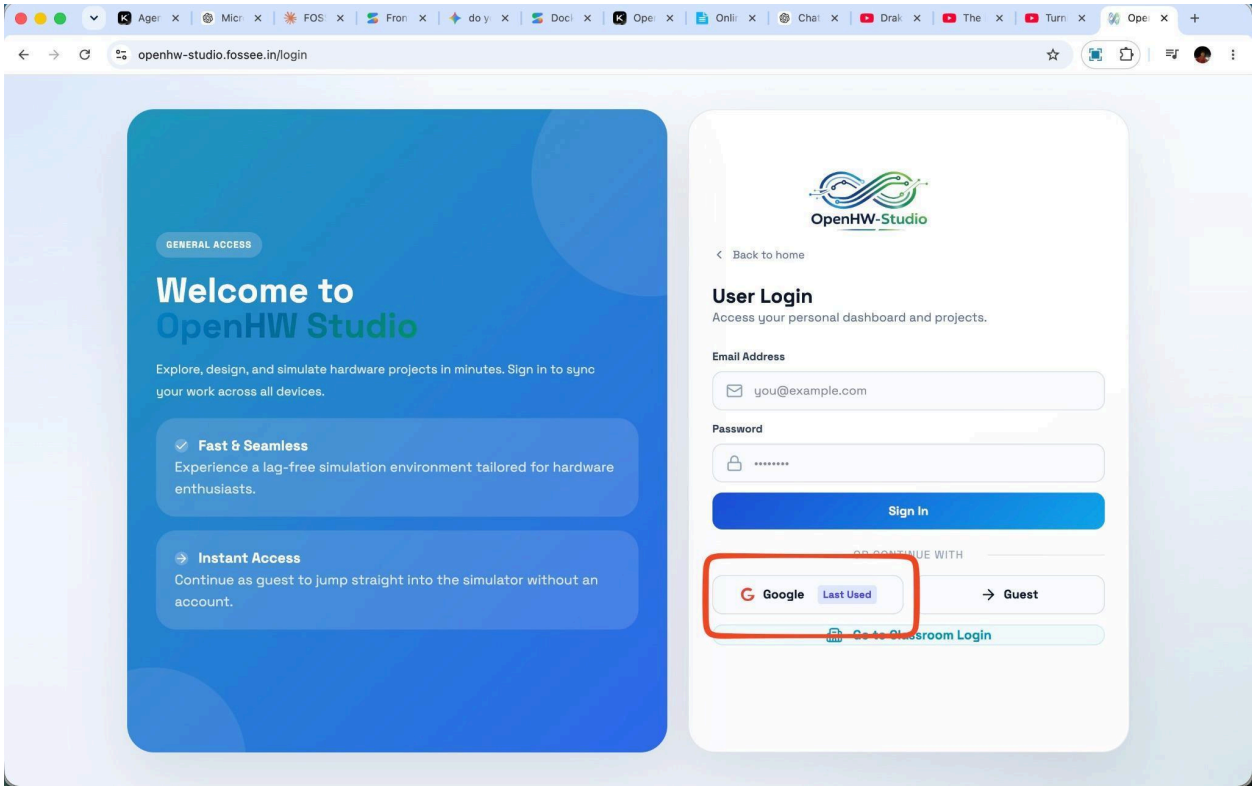


Homepage after layout fixes

The technical specs panel on the right ("16 MHz AVR CLOCK", "60 fps RENDER SPEED") with the circuit graphic was part of stabilizing this layout. The page now communicates what the platform does within seconds of loading.

4.2 Global Theme Persistence & UI Polish

- **Theme Persistence:** I fixed issues where the user's Dark Mode preference was lost upon page refresh. Implemented localStorage theme state persistence and injected CSS overrides so that Authentication pages respected the dark theme immediately on load, rather than flashing white before the React app hydrated.
- **Quality of Life Improvements:** Designed and integrated a "Last Used" badge on the login screen to help students remember their previous authentication method, reducing login friction. This small detail significantly improves the experience for returning users:



Login page showing the "Last Used" badge next to Google Auth

4.3 Example Projects & Link Integrity Audit

I designed and implemented the Examples page at https://openhw-studio.fossee.in/examples, which serves as a browsable catalog of demo projects that users can load directly into the simulator. The page pulls project metadata and presents it in a scannable, visual format — card-based layouts with project names, descriptions, and "Open in Simulator" actions.

Beyond the Examples page, I conducted a comprehensive manual audit of the entire platform for broken links. This wasn't an automated scan — I navigated every page, clicked every link, and verified every resource reference. Issues I found and fixed included:

- **Hardcoded Localhost URLs:** Found and replaced 17 instances across the frontend where service URLs were hardcoded as `localhost:5001`, `localhost:3333`, or `localhost:5000`. These appeared in files like simulatorService.js, execute.ts, and various API utility modules. In development these work, but in any other environment they silently break. I

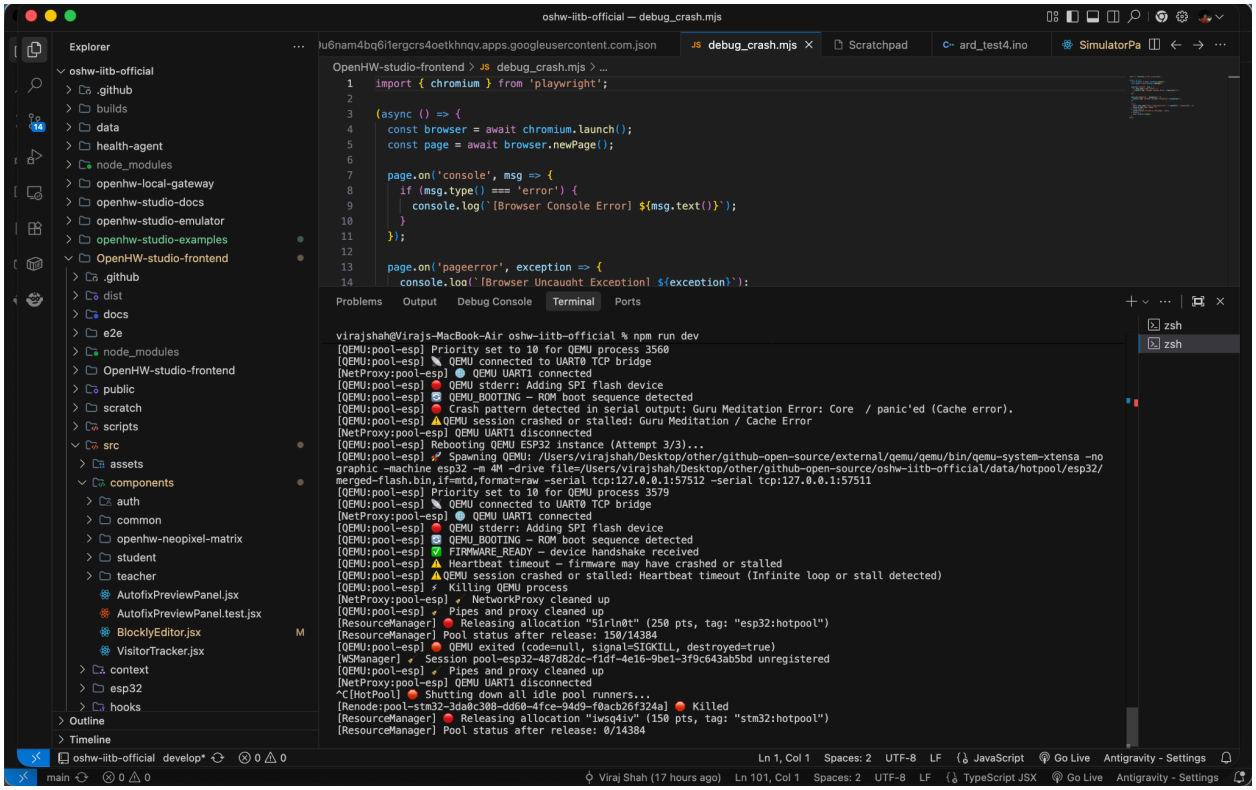
replaced all instances with environment-variable-driven URLs or relative paths that resolve correctly through the Nginx proxy.

- **.exe Local Path References:** Found references to local .exe file paths — remnants of a desktop-app or local-development-era approach where compilation tools were expected to exist on the developer's machine. In a containerized web deployment, these paths are meaningless. I identified these references and ensured the relevant code paths either used the proper worker-based compilation pipeline or were removed.
- **Broken Logo References:** Fixed missing or incorrectly pathed logo images on multiple pages throughout the platform. These were minor but visually jarring — a missing logo on a page immediately signals "this isn't finished."
- **Vite Base Path Configuration:** Verified and adjusted the Vite base configuration so that CSS, JS chunks, images, and the emulator's worker files all resolve correctly in the deployed environment.

4.4 Emulator Component Development

I developed and contributed five hardware emulator components to the openhw-studio-emulator repository: **LED, Diode, Servo, Potentiometer, and Push Button**.

Each component follows the standard three-file structure:



Development log showing QEMU crash and recovery output.

- manifest.json — Pin definitions and metadata aligned with the real hardware's datasheet
- logic.ts — Simulation state machine translating electrical signals (analog voltages, digital HIGH/LOW, PWM duty cycles) into state changes
- ui.tsx — React-based visual representation with real-time feedback and user interaction

- **LED:** Implemented forward voltage drop simulation and current limiting behavior. The UI provides real-time visual feedback showing the LED's illumination state and brightness based on the applied voltage and current.
- **Diode:** Developed simulation logic for forward and reverse bias behavior, including the voltage threshold at which the diode begins conducting. The UI visually indicates the diode's state (conducting vs. blocking).
- **Servo:** Implemented PWM-based angle control logic, translating the duty cycle of the PWM signal (typically 1-2ms pulse width within a 20ms period) to a physical angle (0-180°). The UI displays the servo arm's position in real-time.

- **Potentiometer:** Created analog voltage divider simulation where the slider position maps to an output voltage between 0V and the input voltage. The UI provides an interactive slider for real-time adjustment.
- **Push Button:** Implemented debounced digital input simulation with configurable pull-up and pull-down resistor modes. The UI allows toggling the button state and shows the resulting digital output.

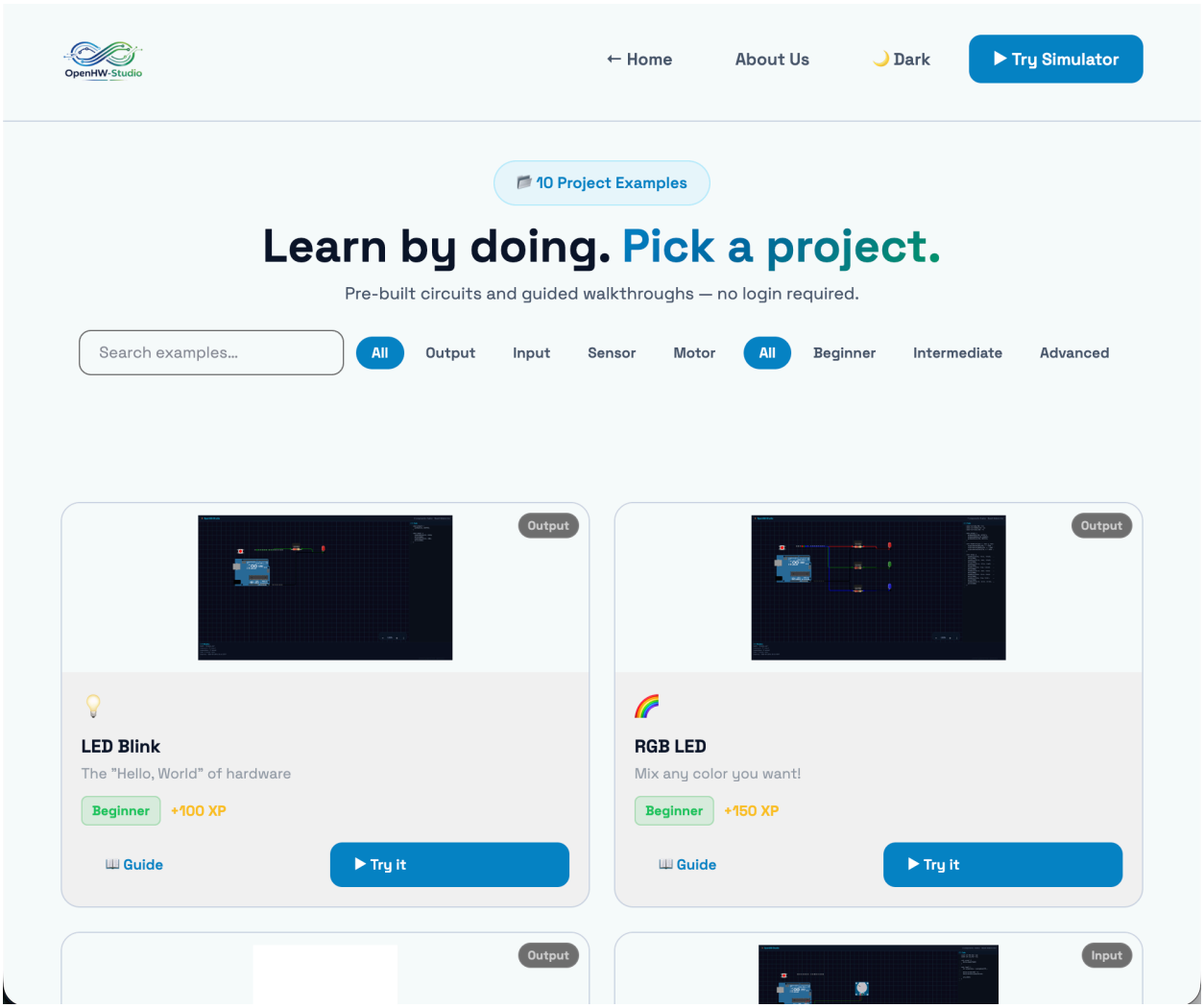
All five components integrated into the existing simulation pipeline without requiring changes to the microcontroller runners or the backend, and were registered in the frontend's component registry to appear as draggable items in the simulator workspace.

4.5 Blockly Educational Content & Example Library

One of my major educational contributions during the internship was expanding the platform's Blockly ecosystem and creating a library of verified example projects that new users can immediately explore and modify.

While OpenHW Studio already supported Blockly-based programming, there was very little curated educational content demonstrating how beginners should use it. Simply providing a visual programming interface is often not enough—new users need working examples that they can study, modify, and experiment with before building their own projects.

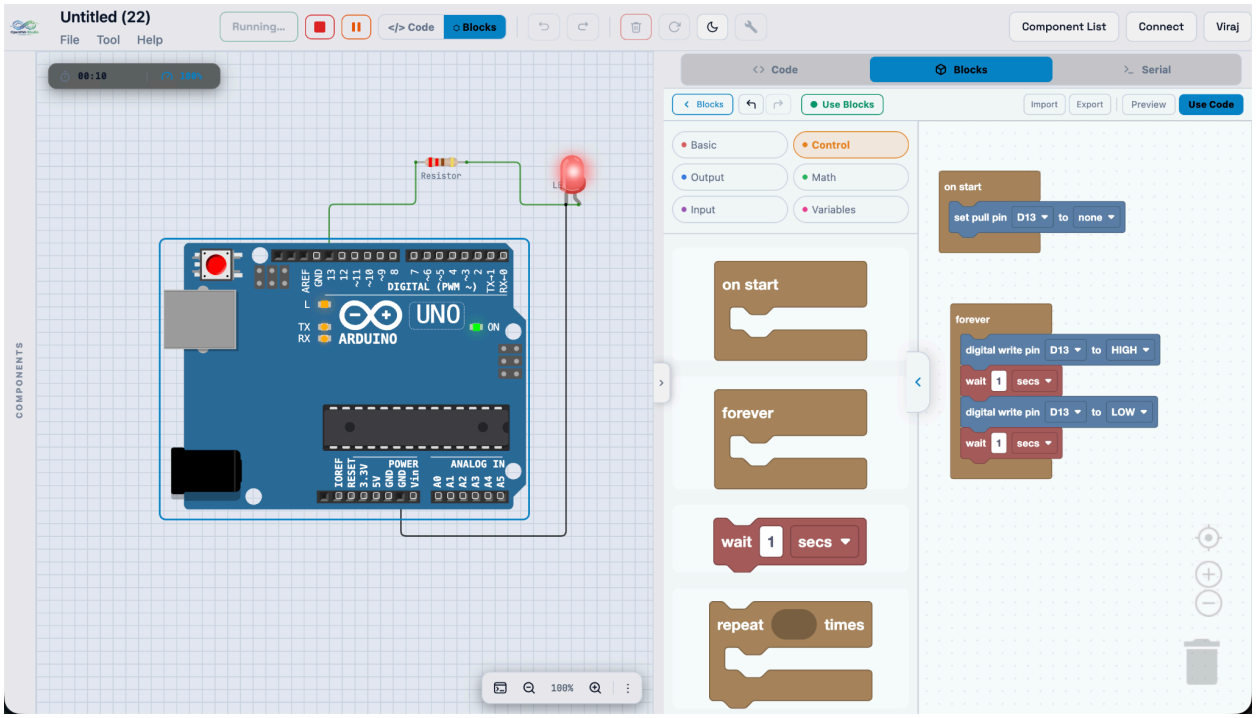
To address this, I designed and implemented the platform's **Examples** page, providing a central location where users can browse and launch ready-to-use hardware demonstrations directly inside the simulator. Each example is intended to demonstrate a specific embedded systems concept, allowing students to move from guided learning towards independent experimentation.



OpenHW Studio Examples page

Unlike conventional source code, Blockly programs are stored as structured XML documents that describe the complete visual block graph. Creating these examples therefore involved manually constructing Blockly XML layouts, ensuring that every block, connection, variable, and field was represented correctly. Since Blockly is highly sensitive to malformed XML, each project had to be carefully validated before it could be included in the platform.

Beyond creating the visual block layouts, I verified that every Blockly example functionally matched the generated Arduino C++ source code. Any inconsistencies between the visual blocks and the generated source code were corrected before the examples were added to the library.



Blockly workspace demonstrating one of the verified educational examples

This verification process ensured that every published example serves as a reliable learning resource rather than simply acting as a demonstration. Students can open an example, inspect both its visual Blockly representation and the generated Arduino code, run the simulation, and then modify the project themselves to better understand how each hardware component behaves.

Together, the Examples page and its verified Blockly projects significantly improve the educational value of OpenHW Studio by providing beginners with practical reference implementations that bridge the gap between visual programming and embedded software development.

4.6 Complete Blockly Visual Programming & Beginner Tour

While developing the Examples page and the surrounding educational workflow, I created and integrated the platform's Blockly-based learning content to make the simulator more accessible to beginners. This included translating selected Arduino logic into Blockly visual programs, validating that the generated code matched the

expected simulator behavior, and ensuring that the examples could be launched and understood without requiring users to start from a blank canvas.

The Blockly content now functions as part of the core educational experience on the platform. Users can explore structured example projects, observe how block-based logic maps to hardware behavior, and transition from visual programming to code-based experimentation in a guided manner. This was especially important for beginners, because it reduced the initial learning barrier while still preserving the depth and flexibility of the simulator.

To achieve this, I completed the following implementations:

1. Complete Blockly-to-C++ Transpilation: I implemented working Blockly code for all example projects on the platform. Previously, some examples lacked a fully functional block-based counterpart. I mapped the existing Arduino C++ logic back into functional Blockly XML graphs, ensuring that the transpiler correctly generates valid C++ for every example without syntax errors or missing logic.
2. Examples Page Integration: I updated the Examples page UI to clearly distinguish between code-based and block-based examples. This allows users to filter and launch Blockly projects directly into the workspace with pre-loaded hardware configurations, closing the gap between browsing and active simulation.
3. Interactive Beginner Tour: I designed and implemented a guided walkthrough tour within the Blockly workspace itself. This tour uses interactive tooltips to teach new users how to drag blocks, connect logic, interact with hardware components, and run their first simulation. This significantly reduces the “blank canvas” intimidation that beginners face when first opening a hardware simulator, guiding them from their first block drop to their first successful LED blink.

In addition to authoring the Blockly examples, I verified that the example library remained consistent with the updated simulator environment and that the block-based workflows were properly linked to the relevant hardware components.

This made Blockly not just an isolated feature, but a practical educational layer integrated into the platform’s existing ecosystem.

Chapter 5: Summary

My contributions to OpenHW Studio established the core infrastructure required to take the project from a localized development tool to a publicly accessible, secure educational platform.

Area	What Was Done	Impact
Dockerization	Wrote Dockerfiles (multi-stage builds) and docker-compose.yml for all 7 services; resolved ARM64/AMD64 mismatches	The entire stack is reproducibly deployable with a single command across architectures
VM Deployment	Configured Nginx reverse proxy, WebSocket forwarding, static file serving; navigated jump host access	Platform runs self-hosted at openhw-studio.fossee.in
Security Audit	Removed 118+ debug logs exposing credentials; patched timing attack; identified DoS/DDoS attack surfaces	Eliminated data leaks and hardened the backend
Endpoint Hardening	Secured compilation and admin routes with authentication middleware	Closed unauthenticated access to resource-intensive endpoints
Google OAuth	Initial implementation of Passport.js OAuth flow; handled edge cases	Streamlined user onboarding
About Us Redesign	Full glassmorphic UI redesign with self-contained stylesheet	Page went from non-functional to polished
Homepage Fix	Repaired broken layout, stabilized hero section and specs panel	First impression of the platform is no longer broken

Area	What Was Done	Impact
Examples Page	Designed and implemented browsable demo project catalog	Users can discover and load example projects
Link Integrity Audit	Manual scan of entire platform; fixed 17 hardcoded URLs, broken logos, .exe references	Platform links and assets work correctly in production
Theme Persistence	Fixed dark mode loss on refresh; injected CSS for auth pages	Consistent visual experience
“Last Used” Badge	Designed and integrated on login page	Reduced login friction for returning users
5 Emulator Components	Built LED, Diode, Servo, Potentiometer, Push Button	Expanded the simulation catalog
CI/CD Groundwork	Architected GitHub Actions workflow for automated builds and deployments	Laid foundation for automated deployment pipeline
Robust Backend Tests	Created 30 core tests to ensure future contributions are safer and follow best-practices.	Allows guardrail based consistent non-destructive contribution
Blockly Educational Content	Designed the Examples page, authored and verified Blockly XML projects, validated generated XML-Arduino code, created tour	Lowered the learning curve by providing reliable, beginner-friendly educational examples

Chapter 6: Conclusion

This internship was a genuinely valuable experience for me, both as a developer and as a person. Working on OpenHW Studio gave me the chance to deal with real problems in a live open-source project — not just writing code for the sake of it, but building, testing, breaking, fixing, and improving something that other people actually use. That process taught me a lot more than feature work alone ever could.

What stood out most to me was how much responsibility came with working in a shared codebase. I had to think beyond whether something worked on my machine and consider whether it was clear, stable, maintainable, and safe for everyone else involved in the project. Along the way, I learned to pay attention to details that are easy to ignore at first — deployment issues, hidden bugs, broken links, security gaps, and the small usability fixes that make a platform feel complete instead of half-finished. Those lessons will stay with me long after this internship.

Completing the Blockly implementation and designing the interactive onboarding tour was particularly rewarding because it required thinking from the perspective of a beginner rather than a developer. It reinforced the importance of creating tools that are not only technically correct, but also approachable and easy to learn from, ensuring that a student's first interaction with embedded systems feels intuitive rather than intimidating.

I am sincerely grateful to Prof. Prabhu Ramchandran, Prof. Rajesh Kushalkar, Prof. Pratik Bhosale, and Prof. Pratik Nemane for their guidance and support throughout the internship. Their feedback helped shape the direction of my work and gave me the confidence to take ownership where it was needed.

I believe OpenHW Studio has strong potential as an accessible learning platform for hardware education. By combining simulation, structured workflows, and a more beginner-friendly experience, it can make embedded systems feel less intimidating for students who do not have access to physical lab equipment. Being part of that effort has been one of the most meaningful parts of my internship.

Looking ahead, this experience has left me with a stronger foundation in full-stack development, deployment, and open-source collaboration, along with a deeper appreciation for building software that is practical, usable, and worth improving over time.

Chapter 7: Future Work

While the foundational infrastructure is now stable and deployed, several areas can be expanded:

1. **Guest Compilation Middleware:**

Complete the implementation of the `protectRouteOrGuest` middleware with appropriate rate limiting, allowing users to compile and simulate projects without an account while protecting the platform against abuse and resource exhaustion.

2. **Expanded Hardware Coverage:**

Continue expanding the simulator's hardware library by adding support for additional components, sensors, communication modules, and microcontroller peripherals to improve the breadth of projects that can be built and simulated within OpenHW Studio.

3. **Complete Blockly & Interactive Tutorial:**

Develop a comprehensive onboarding experience that introduces new users to the Blockly interface through more guided walkthroughs, interactive tutorials, and progressively challenging exercises to improve accessibility for beginners.

4. **Beginner-Friendly Schematic Explainer Videos on *Youtube*:**

Create a beginner-oriented YouTube playlist covering the fundamentals of electronics, circuit design, and the OpenHW Studio workflow, enabling students with little or no hardware background to start using the platform effectively. Possibly publishing it on the FOSSEE Youtube channel (@fosseeiitbombay).

5. **Dedicated Blockly Deep Dive:**

Develop an advanced tutorial series focused entirely on Blockly programming, covering custom blocks, program structure, hardware interaction, debugging techniques, and best practices for building complex embedded applications.

References

1. https://github.com/OpenHW-Studio
2. https://github.com/OpenHW-Studio/OpenHW-studio-frontend
3. https://github.com/OpenHW-Studio/openhw-studio-emulator
4. https://github.com/OpenHW-Studio/openhw-studio-backend
5. https://github.com/OpenHW-Studio/openhw-studio-examples
6. https://github.com/OpenHW-Studio/openhw-studio-docs
7. https://openhw-studio.fossee.in/
8. https://fossee.in/ — FOSSEE, IIT Bombay
9. https://www.youtube.com/channel/UCMtt6exSCmZI7JU73S6Wz_A — FOSSEE Official Youtube Channel

Thank You.