eSim Summer Fellowship 2026

On

Developing alternative KiCad FrontEnd

Submitted by

Jahnvi Verma

Bachelor of Technology Computer Science and Engineering, 4th Year
Vellore Institute of Technology, Bhopal

Under the guidance of

Prof. Prabhu Ramachandran

Principal Investigator
Department of Aerospace Engineering
Indian Institute of Technology Bombay

July 20, 2026

Acknowledgment

I express my sincere gratitude to Prof. Prabhu Ramachandran for providing me with the opportunity to be part of the FOSSEE internship program and for his continued efforts in promoting open-source engineering tool development. His leadership and vision have been instrumental in fostering meaningful student participation in the open-source ecosystem.

I also acknowledge Prof. Kannan M. Moudgalya for his foundational role in establishing and strengthening the FOSSEE initiative. His contributions to open-source education and the development of the FOSSEE fellowship framework have been pivotal in creating the academic and organisational platform through which this internship was undertaken.

My sincere appreciation extends to my mentor, Sumanto Kar, for his continual support, technical guidance, and encouragement throughout the duration of this project. His insights and feedback played a key role in refining ideas, overcoming challenges, and ensuring timely completion of the tasks assigned to me.

I would also like to thank my internal mentors, Mr. Varad Patil and Ms. Shanthi Priya K for their valuable guidance, coordination, and technical inputs during the internship. Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

Working on the KiCad-Alternating-Frontend project has been a rewarding part of my internship, giving me hands-on exposure to open-source tools like eSim-Cloud and KiCad. I gained practical experience in real-world architectural flow, simulation workflows, and frontend integration for EDA tools. This blend of design and development work has sharpened both my technical and problem-solving skills. The learning from this project will continue to shape my academic and professional growth ahead.

I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely interactions at various stages of this work. Their collective efforts ensured smooth workflow, resource accessibility, and effective project execution.

Abstract

eSim is a free and open-source Electronic Design Automation (EDA) tool developed by the FOSSEE team at IIT Bombay, providing an integrated environment for circuit design, simulation, and analysis through tools such as KiCad, Ngspice, GHDL, and Verilator.

This internship project focuses on building a native desktop wrapper for the FOSSEE eSim-Cloud EDA stack using Tauri v2, with the objective of packaging the browser-based eSim-Cloud platform into a standalone, self-contained desktop application, removing the need for users to manually set up and manage the underlying services themselves.

The implementation uses a Tauri v2 shell with a Rust backend and a Vite + TypeScript frontend, served through a native WebView. On first launch, the application automatically clones the FOSSEE/esim-cloud repository and manages its Docker Compose lifecycle, handling Docker and Git detection, backend readiness polling, and progress streaming to the user interface. A screen-transition system, splash screen, and offline error handling were implemented to give the application a native, polished feel.

Multiple integration issues were identified and resolved during development, including PostgreSQL version incompatibilities in the upstream Docker Compose configuration, nginx errors caused by slow frontend cold-builds, and a startup race condition where the UI loaded before the backend was ready. The project originated as a proposal for an independent KiCad-alternative frontend, later pivoted, under mentor guidance, to wrapping the existing eSim-Cloud stack for greater compatibility and maintainability.

The complete implementation and project repositories are publicly available at:
`https://github.com/FOSSEE/eSim_Schematic_Editor`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Electronic Design Automation (EDA) has become an essential component of modern electronics engineering, enabling engineers to design, simulate, verify, and manufacture complex electronic systems with improved accuracy and efficiency. Traditional EDA software provides tools for schematic capture, Printed Circuit Board (PCB) design, circuit simulation, and embedded system development. However, many commercial EDA solutions require expensive licenses, making them inaccessible to many students and educational institutions.

To promote accessible engineering education, the **Free/Libre and Open Source Software for Education (FOSSEE)** project at the **Indian Institute of Technology Bombay (IIT Bombay)** developed **eSim**, a free and open-source Electronic Design Automation platform released under the GNU General Public License (GPL). eSim provides a comprehensive environment for electronic circuit design, simulation, and PCB development without licensing restrictions, making it an important educational tool for engineering students, researchers, and educators.

To improve portability and simplify deployment, eSim was later extended into **eSim-Cloud**, a cloud-native platform built using Docker and Docker Compose. This architecture allows the complete application stack to execute consistently across different operating systems while minimizing dependency-related issues. Although containerization improves maintainability and scalability, it also introduces deployment complexity for users unfamiliar with Docker, Git, and command-line operations. This challenge motivated the development of **esim-desktop**, a native desktop application designed to automate the entire deployment process and provide users with a simple, intuitive interface for accessing the eSim-Cloud platform.

1.2 Overview of eSim-Cloud

eSim-Cloud is the cloud-native implementation of the eSim Electronic Design Automation platform developed under the FOSSEE initiative. Unlike the traditional desktop version, eSim-Cloud follows a containerized architecture in which all application services execute within isolated Docker containers. The platform combines multiple services, including a web-based

EDA editor, Arduino simulation module, PostgreSQL database, backend APIs, and an Nginx reverse proxy, all orchestrated using Docker Compose.

The EDA editor enables users to create electronic schematics, perform PCB layout design, and simulate circuits through a browser interface, while the Arduino module supports embedded system simulation and program execution. The modular architecture ensures that each service can operate independently while communicating efficiently through Docker's internal networking.

For local execution, users traditionally needed to install Docker and Git, clone the official repository, execute Docker Compose commands, and manually monitor the initialization process until all services became operational. In addition to the local deployment, FOSSEE also maintains a hosted instance of eSim-Cloud, allowing users to access the application directly through a web browser when local execution is unavailable.

The **esim-desktop** application developed during this internship functions as an intelligent wrapper around this ecosystem. It automates repository management, container orchestration, backend health verification, and application startup while preserving complete compatibility with the original eSim-Cloud architecture.

1.3 Objectives of the Project

The primary objective of this project was to simplify the deployment and execution of the eSim-Cloud platform by eliminating the technical complexity associated with Docker and command-line operations. The application was designed to provide a native desktop experience that allows users to launch the complete cloud environment with minimal manual intervention while maintaining compatibility with the official FOSSEE repository.

To achieve this objective, several engineering goals were established. The application automatically verifies the availability of Git and Docker before startup, clones the required repository during the first execution, manages the complete Docker Compose lifecycle, and continuously monitors backend readiness before loading the application interface. Another important objective was to improve the user experience by replacing lengthy Docker logs with concise, human-readable progress messages displayed through a graphical splash screen.

The project also focuses on improving reliability by detecting common failure scenarios such as missing software dependencies, inactive Docker services, interrupted network connectivity, and partially cloned repositories. Furthermore, the application automatically shuts down all Docker containers when the desktop window is closed, preventing unnecessary resource consumption and ensuring proper cleanup of background services.

Together, these objectives create a robust and user-friendly deployment solution that significantly improves accessibility for students and educators using the eSim platform.

1.4 Key Engineering Contributions

The development of **esim-desktop** introduced several engineering contributions that enhance both the usability and reliability of the eSim-Cloud ecosystem. The most significant contribution

is the implementation of a zero-configuration deployment workflow that automates repository cloning, Docker Compose execution, backend monitoring, and application loading without requiring direct user interaction with the terminal.

Another major contribution is the implementation of a real-time log aggregation mechanism that converts verbose Docker and build logs into meaningful progress updates. This greatly improves the user experience by providing clear feedback during lengthy initialization processes. The application also incorporates a multi-stage backend health verification mechanism that validates network connectivity, Docker container status, and application-level HTTP responses before loading the embedded interface. This approach eliminates premature application loading and significantly reduces HTTP 502 gateway errors during cold starts.

From a software engineering perspective, the project demonstrates the effective integration of Rust, Tauri v2, TypeScript, Docker, and asynchronous programming using Tokio into a unified desktop application. The implementation of safe workspace management, deterministic application lifecycle handling, and automatic Docker cleanup further improves system reliability while ensuring compatibility with future updates of the upstream eSim-Cloud repository.

Methodology Overview The development of **esim-desktop** followed a structured and iterative software engineering methodology consisting of multiple implementation phases. Initially, the architecture of the existing eSim-Cloud platform was studied to understand its deployment workflow, Docker Compose configuration, service dependencies, and startup sequence. This analysis helped identify the major usability challenges encountered by users during local installation.

Following the requirement analysis, the project was scaffolded using Tauri v2 with a TypeScript frontend and a Rust backend. Individual modules responsible for dependency verification, repository management, Docker orchestration, health monitoring, and frontend communication were developed independently before being integrated into the complete application. Continuous testing was performed after each implementation stage to ensure correct interaction between the frontend interface and backend services.

Several engineering issues encountered during development—including Docker startup synchronization, PostgreSQL compatibility, backend readiness detection, Git branch configuration, and plugin permission management—were resolved through iterative debugging and architectural refinement. Finally, the completed application underwent comprehensive validation across multiple deployment scenarios, including fresh installations, dependency failures, network interruptions, and repeated application launches. This systematic methodology ensured that the final application remained stable, reliable, and suitable for production use within the FOSSEE ecosystem.

1.5 Organisation of the Report

This report is organized into nine chapters, each describing a specific aspect of the project development lifecycle. Chapter 1 introduces the project background, objectives, engineering contributions, and overall methodology. Chapter 2 presents a detailed survey of the technologies used during development, including Tauri, Rust, Docker, Docker Compose, and the eSim-Cloud ecosystem.

Chapter 3 defines the problem statement and discusses the motivation behind developing a native desktop wrapper for eSim-Cloud. Chapter 4 explains the complete system architecture, including the finite state machine governing application startup, Docker orchestration, log aggregation, and backend health verification mechanisms. Chapters 5 and 6 describe the implementation details of the frontend and backend respectively, covering important modules, communication mechanisms, and software design decisions.

Chapter 7 discusses the major engineering challenges encountered during development and explains the solutions adopted to overcome them. Chapter 8 presents the testing methodology and validation results obtained under different deployment scenarios. Finally, Chapter 9 summarizes the project outcomes, discusses existing limitations, and proposes future enhancements that can further improve the functionality and usability of the application. The report concludes with references and an appendix containing the internship work log and supporting documentation.

Chapter 2

Background and Technology Survey

2.1 Desktop Application Frameworks: Electron vs. Tauri

Modern desktop applications are increasingly developed using web technologies, allowing developers to build cross-platform software from a single codebase. Among the most popular frameworks for this purpose are **Electron** and **Tauri**, both of which enable developers to create native desktop applications using familiar frontend technologies such as HTML, CSS, and JavaScript. Although both frameworks provide cross-platform compatibility, they differ significantly in architecture, performance, resource consumption, and deployment strategy.

Electron is one of the earliest and most widely adopted frameworks for cross-platform desktop application development. It combines the Chromium browser engine with a bundled Node.js runtime, enabling developers to execute web applications inside a desktop environment. Applications such as Visual Studio Code, Discord, Slack, and Microsoft Teams are built using Electron because of its mature ecosystem and extensive library support.

However, since every Electron application packages an independent copy of Chromium and Node.js, the resulting application binaries are often very large, typically exceeding 100 MB. Additionally, Electron applications consume substantial system memory even during idle execution, making them less suitable for lightweight engineering tools targeted toward students using resource-constrained computers.

Tauri adopts a fundamentally different architectural approach. Instead of bundling Chromium, Tauri utilizes the operating system's native WebView technology, such as WebView2 on Windows, WebKit on macOS, and WebKitGTK on Linux. The backend is implemented in Rust, providing excellent performance, low memory consumption, and strong memory safety guarantees. Since the rendering engine already exists within the operating system, the compiled application remains significantly smaller while requiring fewer runtime resources.

For the **esim-desktop** project, Tauri was selected because the application primarily serves as a native wrapper around the existing web-based eSim-Cloud interface. The desktop application displays a splash screen, error pages, and an embedded WebView rather than rendering complex custom user interfaces. Consequently, bundling an entire Chromium instance would unnecessarily increase application size without providing additional functional benefits. Tauri's lightweight architecture, faster startup time, lower memory footprint, and seamless integration with Rust made it the ideal framework for implementing a high-performance desktop launcher capable

of orchestrating Docker containers and communicating with the underlying operating system efficiently. This decision aligns well with the project's objective of delivering a fast, portable, and user-friendly desktop application for educational environments.

Table 2.1: Comparison between Electron and Tauri

Feature	**Electron**	**Tauri**
Rendering Engine	Bundled Chromium	Native Operating System WebView
Backend Runtime	Node.js	Rust
Installer Size	Typically 100–200 MB	Usually below 20 MB
Memory Consumption	High	Low
Startup Performance	Moderate	Fast
Security	Larger attack surface	Rust memory safety and capability-based security
Best Use Case	Large web applications	Lightweight native desktop applications
Selected for this Project	No	Yes

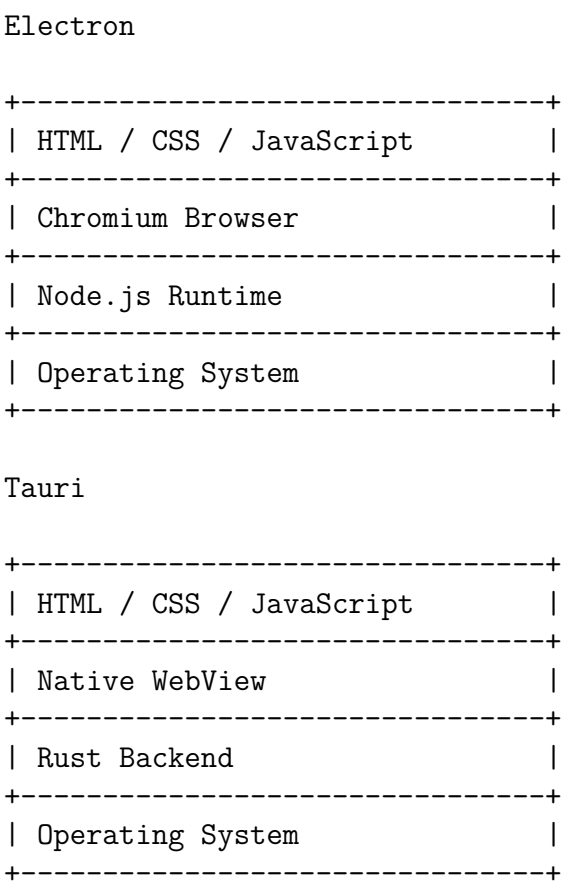

Figure 2.1: Desktop Framework Architecture

2.2 Containerisation with Docker and Docker Compose

Containerization has become one of the most significant advancements in modern software deployment, enabling applications to execute consistently across different operating systems while eliminating dependency-related conflicts. Unlike traditional software installation, where applications rely on libraries and runtime environments already present on the host system, containerization packages an application together with all its dependencies, configuration files, and runtime components into a single isolated unit called a **container**.

This isolation ensures that applications behave identically regardless of the underlying hardware or operating system, thereby improving portability, reproducibility, and maintainability.

Docker is the most widely adopted containerization platform and forms the foundation of the **eSim-Cloud** architecture. Rather than installing numerous software packages manually, Docker creates lightweight containers that share the host operating system's kernel while maintaining isolated execution environments. Since containers require significantly fewer resources than traditional virtual machines, they offer faster startup times, lower memory consumption, and improved deployment efficiency.

While Docker manages individual containers, **Docker Compose** provides a higher-level orchestration mechanism for applications composed of multiple interconnected services. Using a declarative YAML configuration file, Docker Compose defines application components such as databases, web servers, APIs, reverse proxies, and networking configurations. A single command, `docker compose up`, automatically creates networks, initializes containers, establishes dependencies, and starts all required services in the appropriate sequence.

The eSim-Cloud platform utilizes Docker Compose to orchestrate multiple services including the EDA frontend, Arduino simulator, PostgreSQL database, backend APIs, and Nginx reverse proxy. Although Docker Compose greatly simplifies infrastructure deployment, it primarily targets developers and system administrators. During initialization, it generates verbose terminal logs that are difficult for non-technical users to interpret and provides no indication of overall startup progress or application readiness.

To overcome these limitations, **esim-desktop** introduces an intelligent orchestration layer that monitors Docker Compose execution in real time. Instead of exposing raw container logs, the application translates technical output into concise, human-readable status messages displayed through a graphical splash screen. Additionally, a multi-stage health polling mechanism verifies that all backend services are fully operational before loading the application interface. This enhancement significantly improves the usability of Docker-based deployments while preserving the robustness and portability offered by containerization.

Table 2.2: Advantages of Docker-Based Deployment

Feature	Traditional Installation	Docker-Based Deployment
Dependency Management	Manual	Automatic
Environment Consistency	Low	High
Cross-Platform Compatibility	Limited	Excellent
Installation Complexity	High	Low
Isolation	Minimal	Complete
Scalability	Moderate	High
Maintenance	Difficult	Simplified

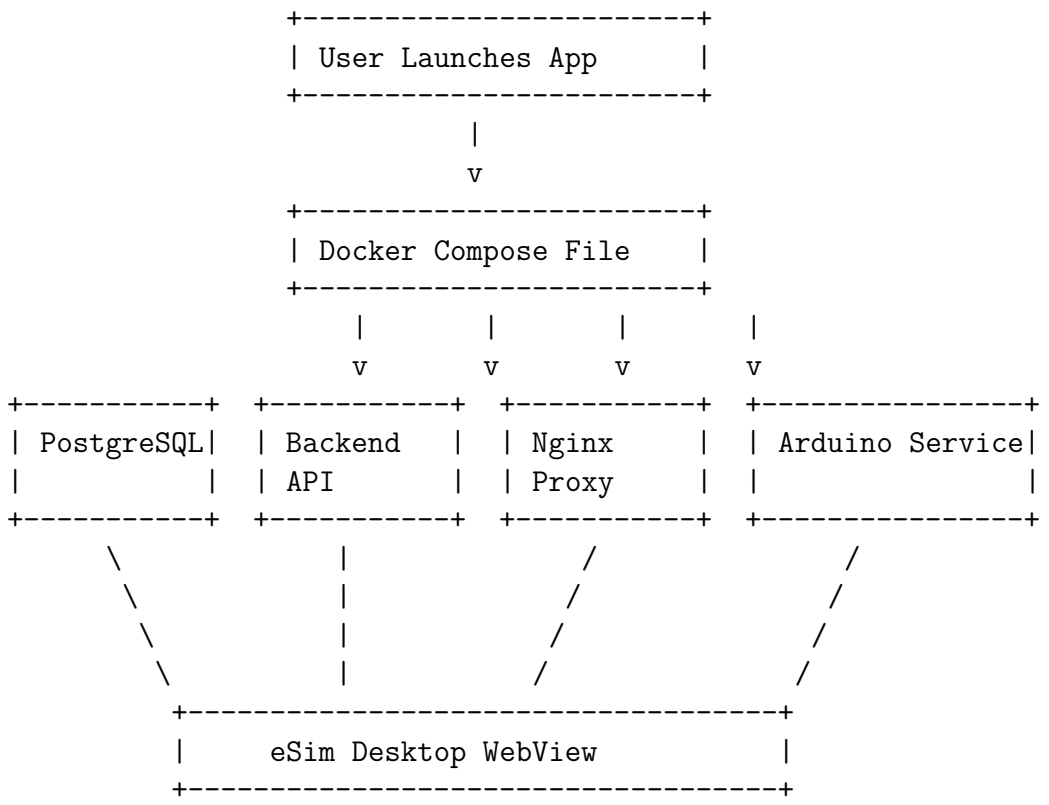

Figure 2.2: Docker Compose Workflow

2.3 eSim-Cloud and the FOSSEE Ecosystem

The **Free/Libre and Open Source Software for Education (FOSSEE)** project, initiated at the **Indian Institute of Technology Bombay (IIT Bombay)**, promotes the adoption and development of open-source software for engineering education. The project encourages educational institutions to replace proprietary software with freely available alternatives that provide comparable functionality while eliminating licensing costs. Through continuous community contributions, documentation, workshops, and software development initiatives, FOSSEE has established a comprehensive ecosystem supporting scientific computing, simulation, programming, computer-aided design, and electronic system development.

One of the major projects under the FOSSEE initiative is **eSim**, an open-source Electronic

Design Automation (EDA) platform developed specifically for circuit design, simulation, PCB layout, and embedded systems education. eSim integrates multiple engineering tools into a unified environment, allowing users to perform schematic capture, SPICE simulation, PCB routing, and Arduino programming without relying on proprietary software. Its open-source nature enables continuous improvements from contributors while making advanced engineering software accessible to students and researchers worldwide.

To modernize software deployment, eSim evolved into **eSim-Cloud**, a browser-based platform built upon containerized microservices. The platform packages all application components—including frontend interfaces, backend services, databases, and reverse proxies—inside Docker containers managed using Docker Compose. This architecture simplifies deployment, improves portability, and ensures consistent execution across different computing environments.

Rather than modifying the upstream project, **esim-desktop** directly interacts with the official **FOSSEE/esim-cloud** Git repository. During its initial execution, the application automatically clones the latest repository, prepares the local workspace, and launches the Docker Compose stack. By relying entirely on the upstream repository instead of maintaining a separate fork, the application remains compatible with future improvements released by the FOSSEE development team. This design significantly reduces maintenance overhead while ensuring that users always benefit from the latest updates, bug fixes, and feature enhancements contributed to the official eSim ecosystem.

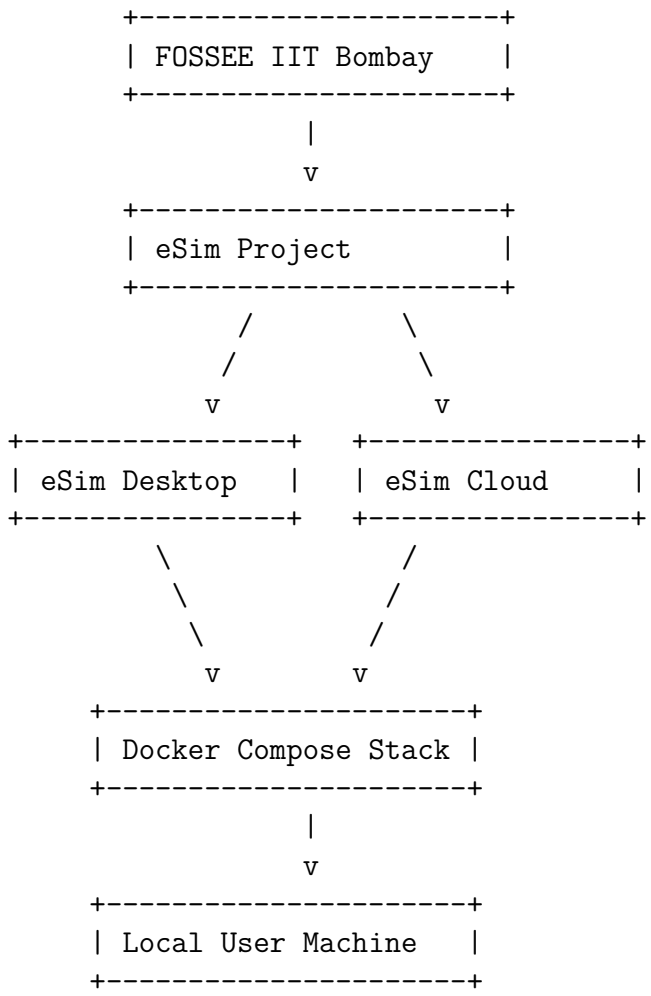

Figure 2.3: Position of eSim-Desktop within the FOSSEE Ecosystem

2.4 Rust and Tokio for Systems Programming

The backend of **esim-desktop** is implemented entirely in **Rust**, a modern systems programming language designed to provide high performance without compromising memory safety. Unlike traditional systems programming languages such as C and C++, Rust employs a strict ownership and borrowing model that eliminates common programming errors including null pointer dereferencing, memory leaks, race conditions, and buffer overflows during compile time. These characteristics make Rust particularly suitable for applications responsible for filesystem operations, process management, network communication, and asynchronous execution.

Rust also serves as the native backend language of the **Tauri v2** framework, making it a natural choice for implementing platform-specific functionality. Within this project, Rust is responsible for validating Docker and Git installations, managing repository cloning, executing Docker Compose commands, monitoring backend services, handling filesystem operations, and communicating with the frontend through Tauri's Inter-Process Communication (IPC) mechanism.

To ensure that long-running operations do not block the graphical user interface, the project utilizes **Tokio**, Rust's asynchronous runtime. Tokio enables the application to execute multiple tasks concurrently using asynchronous programming primitives such as futures and asynchronous functions. Operations including Docker startup, Git repository cloning, TCP socket polling, HTTP health verification, and log monitoring are executed independently of the UI thread, ensuring that the desktop interface remains responsive even during lengthy initialization procedures.

Additionally, Tokio's timeout functionality is extensively used throughout the application to prevent indefinite blocking caused by network delays or stalled Docker containers. This asynchronous architecture significantly improves application responsiveness while providing reliable execution of concurrent background tasks. The combination of Rust's safety guarantees and Tokio's efficient task scheduling forms the foundation of a robust backend capable of managing complex containerized workflows without sacrificing performance or reliability.

Table 2.3: Role of Rust and Tokio

Component	Responsibility
Rust	Backend implementation
Ownership Model	Memory safety and ownership management
Tauri	Desktop application backend
Tokio	Asynchronous runtime for concurrent task execution
Async Tasks	Docker, Git, and HTTP polling operations
Timeout API	Prevent indefinite waiting during asynchronous operations
IPC	Frontend–backend communication between WebView and Rust backend

2.5 Existing Tools and Limitations

Prior to the development of **esim-desktop**, users wishing to execute eSim-Cloud locally were required to follow a completely manual deployment workflow. This process involved installing Docker Desktop, configuring Git, cloning the official repository, navigating through terminal commands, executing Docker Compose, monitoring extensive console logs, and manually determining when backend services had become operational. Although technically straightforward for experienced developers, this workflow presented considerable challenges for students and educators with limited experience in containerized software deployment.

One of the major limitations of the traditional approach was the complete absence of graphical feedback. Docker Compose outputs highly detailed technical logs intended primarily for debugging rather than user interaction. During the first startup, image downloads, dependency installation, frontend compilation, and database initialization may take several minutes, often leading users to assume that the application has become unresponsive. Furthermore, Docker reports containers as "running" even when internal application services are still compiling, resulting in premature browser access and HTTP 502 gateway errors.

Another limitation involved lifecycle management. After completing their work, users needed to manually execute `docker compose down` to terminate all running containers. Failure to perform this step resulted in unnecessary resource consumption, background processes, and potential port conflicts during subsequent executions.

The **esim-desktop** application addresses each of these limitations through an integrated desktop interface that automates repository management, Docker lifecycle control, backend health verification, progress reporting, and graceful shutdown. By abstracting infrastructure complexity behind a graphical interface, the application transforms a technically demanding deployment process into a user-friendly experience while preserving complete compatibility with the official eSim-Cloud project.

Table 2.4: Comparison of Existing Workflow and Proposed Solution

Feature	**Traditional Workflow**	**esim-desktop**
Manual Git Clone	Required	Automated
Docker Compose Commands	Manual	Automatic
Health Verification	Not Available	Multi-stage
Progress Tracking	Raw Terminal Logs	Graphical Progress Screen
Repository Management	Manual	Automatic
Error Handling	Technical Console Errors	User-Friendly Messages
Container Shutdown	Manual	Automatic
User Experience	Developer-Oriented	Beginner-Friendly

Chapter 3

Problem Statement and Proposed Approach

3.1 Problem Statement

The evolution of cloud-native software architectures has significantly improved application portability, scalability, and deployment consistency. By leveraging technologies such as Docker and Docker Compose, complex software systems can now be packaged together with their dependencies, allowing identical execution across different operating systems. Although this architectural approach provides substantial benefits for developers and system administrators, it introduces considerable complexity for end users who simply wish to use the application without understanding the underlying infrastructure.

The **eSim-Cloud** platform developed under the FOSSEE initiative follows a fully containerized deployment model. Instead of installing application components directly onto the host operating system, the platform consists of multiple interconnected Docker containers that collectively provide schematic capture, PCB design, Arduino simulation, backend APIs, database services, and reverse proxy functionality. Before accessing the application locally, users must complete several prerequisite steps including installing Docker Desktop, configuring Git, cloning the official repository, executing Docker Compose commands, monitoring container initialization, and determining when the application has become fully operational.

While this workflow is manageable for software developers familiar with containerized applications, it presents a significant usability challenge for the primary target audience of eSim, namely undergraduate engineering students, educators, researchers, and first-time users. Most users are interested in designing and simulating electronic circuits rather than learning Docker commands or debugging container startup failures. Consequently, even minor configuration issues such as inactive Docker services, missing software dependencies, interrupted network connections, or partially cloned repositories can prevent successful application startup.

Another important limitation is the absence of meaningful feedback during application initialization. Docker Compose generates extensive technical logs intended primarily for debugging purposes. During the first execution, containers may spend several minutes downloading images, installing dependencies, initializing databases, and compiling frontend assets. Throughout this process, users receive little indication of actual progress and often assume that the application

has frozen or encountered an error.

Furthermore, Docker considers a container to be "running" immediately after its process starts, even though internal web applications may still be compiling. As a result, users frequently attempt to access the application before backend services become fully available, leading to HTTP 502 gateway errors or incomplete page rendering. The manual deployment workflow also requires users to remember to terminate Docker containers after completing their work. Failure to execute `docker compose down` leaves multiple background services running, unnecessarily consuming CPU resources, memory, storage, and network ports.

These limitations collectively reduce the accessibility of eSim-Cloud despite its technically robust architecture. Therefore, there exists a need for an intelligent desktop application capable of automating deployment, simplifying infrastructure management, and providing a user-friendly interface that abstracts the underlying containerized environment.

Table 3.1: Existing Problems in the Manual Workflow

Problem	**Description**	**Impact**
Manual Installation	Users manually install Docker and Git	Difficult for beginners
Repository Cloning	Requires Git commands	Error-prone
Docker Compose Execution	Terminal-based workflow	Steep learning curve
Startup Monitoring	Raw Docker logs	Poor user experience
Health Verification	No readiness validation	HTTP 502 errors
Error Diagnosis	Technical console messages	Difficult debugging
Shutdown Process	Manual `docker compose down`	Resource wastage
Dependency Management	Multiple prerequisites	Increased setup time

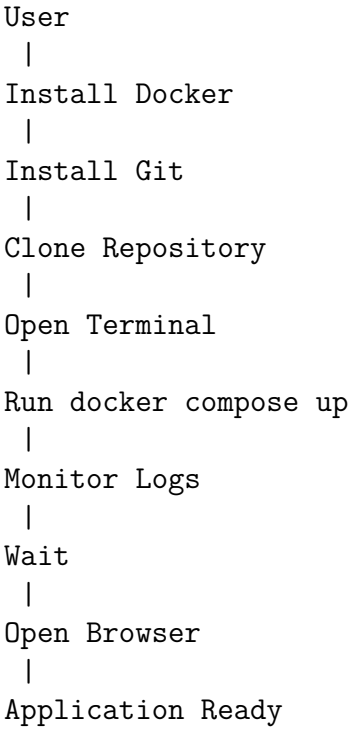

Figure 3.1: Traditional Deployment Workflow

3.2 Functional Requirements

Based on the identified limitations, the proposed desktop application was designed to satisfy several functional requirements that collectively simplify the deployment process while preserving complete compatibility with the original eSim-Cloud ecosystem.

The application must first verify the availability of essential software dependencies including Docker and Git before attempting any deployment operation. If either dependency is missing or incorrectly configured, the application should immediately notify the user through an informative graphical interface rather than displaying low-level system errors.

During the first execution, the application should automatically clone the official FOSSEE eSim-Cloud repository into a dedicated workspace without requiring user interaction with the command line. If the repository already exists, unnecessary cloning operations should be skipped to reduce startup time.

The application must also automate Docker Compose lifecycle management, including container startup, health verification, and graceful shutdown. Throughout the startup process, technical Docker logs should be translated into concise progress messages suitable for non-technical users. Once backend services become fully operational, the application should automatically load the embedded web interface.

Finally, when the desktop application closes, all Docker containers started during execution should be terminated automatically to prevent orphaned processes and unnecessary resource consumption.

Table 3.2: Functional Requirements

Requirement ID	Functional Requirement
FR-01	Verify Docker installation
FR-02	Verify Git installation
FR-03	Clone repository automatically
FR-04	Detect existing workspace
FR-05	Start Docker Compose services
FR-06	Monitor backend health
FR-07	Display graphical progress updates
FR-08	Load embedded eSim interface
FR-09	Handle common failure scenarios
FR-10	Automatically terminate containers

3.3 Non-Functional Requirements

In addition to functional capabilities, the system was designed to satisfy several non-functional requirements that directly influence overall software quality. Performance is an important consideration because users expect the application to launch efficiently without introducing unnecessary delays beyond those required for Docker initialization. Reliability is equally

important since unexpected failures during deployment may leave repositories in inconsistent states or containers partially initialized.

The application should remain portable across major desktop operating systems supported by Tauri while maintaining compatibility with future updates to the upstream eSim-Cloud repository. Maintainability was also considered by separating frontend and backend responsibilities into modular components with clearly defined interfaces.

Usability represents another major non-functional requirement. Since the intended users include students with limited experience in containerized software deployment, the graphical interface should minimize technical complexity while presenting meaningful progress information and actionable error messages.

Table 3.3: Non-Functional Requirements

Category	**Description**
Performance	Fast startup with minimal overhead
Reliability	Stable Docker lifecycle management
Maintainability	Modular software architecture
Portability	Windows, Linux, macOS compatibility
Security	Safe filesystem operations
Scalability	Support future platform enhancements
Usability	Beginner-friendly graphical interface

3.4 Proposed Approach

To address the limitations of the traditional deployment workflow, this project introduces **esim-desktop**, a native desktop application that functions as an intelligent orchestration layer between the user and the eSim-Cloud infrastructure. Rather than modifying the upstream eSim project, the application automates every infrastructure-related operation while preserving the original architecture.

When launched, the application first validates system prerequisites by checking whether Docker and Git are correctly installed. If validation succeeds, it prepares the local workspace and clones the official repository when necessary. The backend then evaluates the current Docker environment to determine whether containers are already running or need to be started. During container initialization, Docker output is continuously processed and converted into human-readable progress messages displayed through a graphical splash screen.

Unlike conventional deployment workflows, the application employs a multi-stage backend health verification mechanism. Instead of assuming readiness immediately after containers start, it validates network connectivity, Docker container status, and application-level HTTP responses before loading the embedded web interface. This significantly improves reliability by preventing premature application startup.

Upon successful validation, the desktop application seamlessly loads the local eSim-Cloud interface inside a native WebView, providing users with the experience of a conventional desktop

application while retaining the flexibility of the underlying web-based platform.

When the user exits the application, all Docker services initiated during execution are terminated automatically, ensuring proper cleanup and efficient resource management.

```
User
 |
Launch eSim-Desktop
 |
Verify Docker
 |
Verify Git
 |
Clone Repository (if needed)
 |
Validate Containers
 |
Start Docker Compose
 |
Backend Health Check
 |
Load eSim Interface
 |
Automatic Shutdown on Exit
```

Figure 3.2: Proposed Automated Workflow

3.5 Chapter Summary

This chapter identified the limitations associated with the traditional deployment process of eSim-Cloud and established the motivation for developing a native desktop orchestration application. The major challenges—including complex installation procedures, dependency management, inadequate startup feedback, lack of backend readiness verification, and manual lifecycle management—were analyzed in detail. Based on these observations, the functional and non-functional requirements of the proposed system were defined. Finally, the overall solution approach adopted in **esim-desktop** was presented, illustrating how automated dependency verification, repository management, Docker orchestration, health monitoring, and graphical progress reporting collectively simplify the deployment experience while maintaining compatibility with the existing FOSSEE ecosystem.

Chapter 4

System Design and Architecture

4.1 System Overview

The architecture of esim-desktop is fundamentally designed around the concept of a native wrapper. Rather than rewriting the eSim-Cloud application for desktop environments, this system acts as an intelligent, automated middleware layer that bridges the gap between the host operating system and a containerized web application. The high-level workflow begins when a user launches the application; the system automatically detects prerequisites, prepares the local workspace, orchestrates the underlying Docker infrastructure, and finally embeds the resulting web application within a native desktop window.

This architecture was chosen to remain completely non-intrusive with respect to the upstream FOSSEE/esim-cloud codebase. By cloning and executing the upstream repository unmodified, all orchestration logic remains securely isolated within the esim-desktop application shell. This ensures that the desktop application inherently maintains compatibility with future updates to the eSim-Cloud platform while shielding non-technical users from the complexities of command-line operations.

4.2 Overall System Architecture

The system operates across four primary conceptual layers: the Frontend, the Backend, the Infrastructure (Docker), and the Payload (eSim-Cloud).

- **Frontend:** Constructed using vanilla TypeScript and bundled via Vite, the frontend is deliberately lightweight. It handles user interface state transitions, renders system-themed splash screens, and houses the embedded iframe that ultimately displays the EDA tool.
- **Backend:** Developed in Rust using the Tauri v2 framework, the backend is responsible for all system-level operations. It interfaces with the host operating system to execute subprocesses, manages file system paths securely, and runs asynchronous tasks using the Tokio runtime.
- **Docker:** Serves as the infrastructural engine. The backend issues programmatic commands to the Docker CLI and Docker Compose to build, start, and stop the required container

stack.

- **eSim-Cloud:** The actual payload application, consisting of EDA and Arduino services, PostgreSQL databases, and Nginx reverse proxies running inside the isolated containers.

Communication between the frontend and backend strictly crosses Tauri's Inter-Process Communication (IPC) boundary.

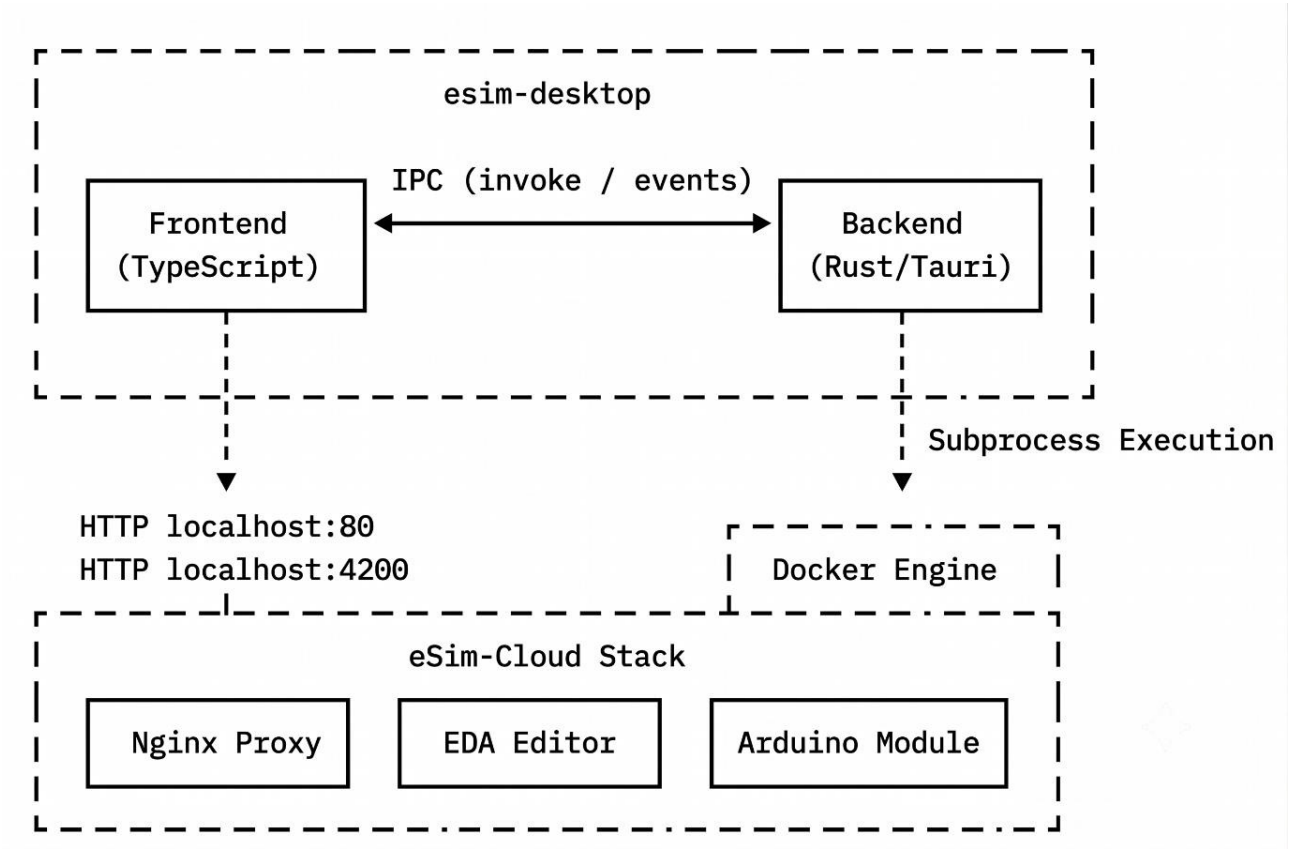

Figure 4.1: Overall Architecture Diagram

4.3 Application Startup Lifecycle

The complexity of configuring an environment, handling missing dependencies, and orchestrating containers is managed by defining the application's startup lifecycle as a deterministic finite state machine (FSM). This formal mathematical model ensures the frontend always renders exactly one correct view without ambiguity.

Let Q be the finite set of states the startup sequence can occupy: $Q = \{Init, CheckDocker, CheckGit, Cloning, ValidateContainers, ComposeUp, PollHealth, Ready, ErrDockerMissing, ErrDaemonDown, ErrGitMissing, ErrCloneFailed\}$

The sequence operates as a deterministic finite automaton $(Q, \Sigma, \delta, q_0, F)$ where the initial state is $q_0 = Init$, the alphabet Σ represents outcome events emitted by system checks (e.g., pass, fail-missing), and F represents terminal states where the application either becomes usable or requires manual user intervention.

Table 4.1: Transition Table for the Startup State Machine

Current State	Event	Next State
Init	(auto)	CheckDocker
CheckDocker	pass	CheckGit
CheckDocker	fail-missing	ErrDockerMissing
CheckDocker	fail-daemon	ErrDaemonDown
CheckGit	pass	Cloning
CheckGit	fail-missing	ErrGitMissing
Cloning	pass / already-cloned	ValidateContainers
Cloning	fail-network	ErrCloneFailed
ValidateContainers	all-running	PollHealth
ValidateContainers	not-all-running	ComposeUp
ComposeUp	pass	PollHealth
PollHealth	healthy	Ready

4.4 Workspace Preparation and Repository Management

A critical requirement for local execution is securing a copy of the FOSSEE repository. On first launch, the `clone.rs` module resolves the user's home directory and safely targets `.esim-desktop/esim-cloud` as the workspace location.

The application utilizes `std::process::Command` to invoke the system's native Git binary directly rather than bundling a separate Rust Git library. This ensures the application inherently respects any custom network proxies or SSH authentication rules the user has already configured globally on their machine.

If the application detects an existing repository folder, the cloning phase is marked as idempotent and bypassed entirely to accelerate warm launches. However, if a clone operation fails mid-execution (such as during a network outage), the system invokes a `validate_cleanup_path` function. This safety mechanism strictly verifies that the target path is a genuine descendant of the application's isolated directory before executing destructive operations, thereby preventing accidental deletion of unrelated user files. The corrupted partial clone is safely purged, ensuring the next retry starts cleanly.

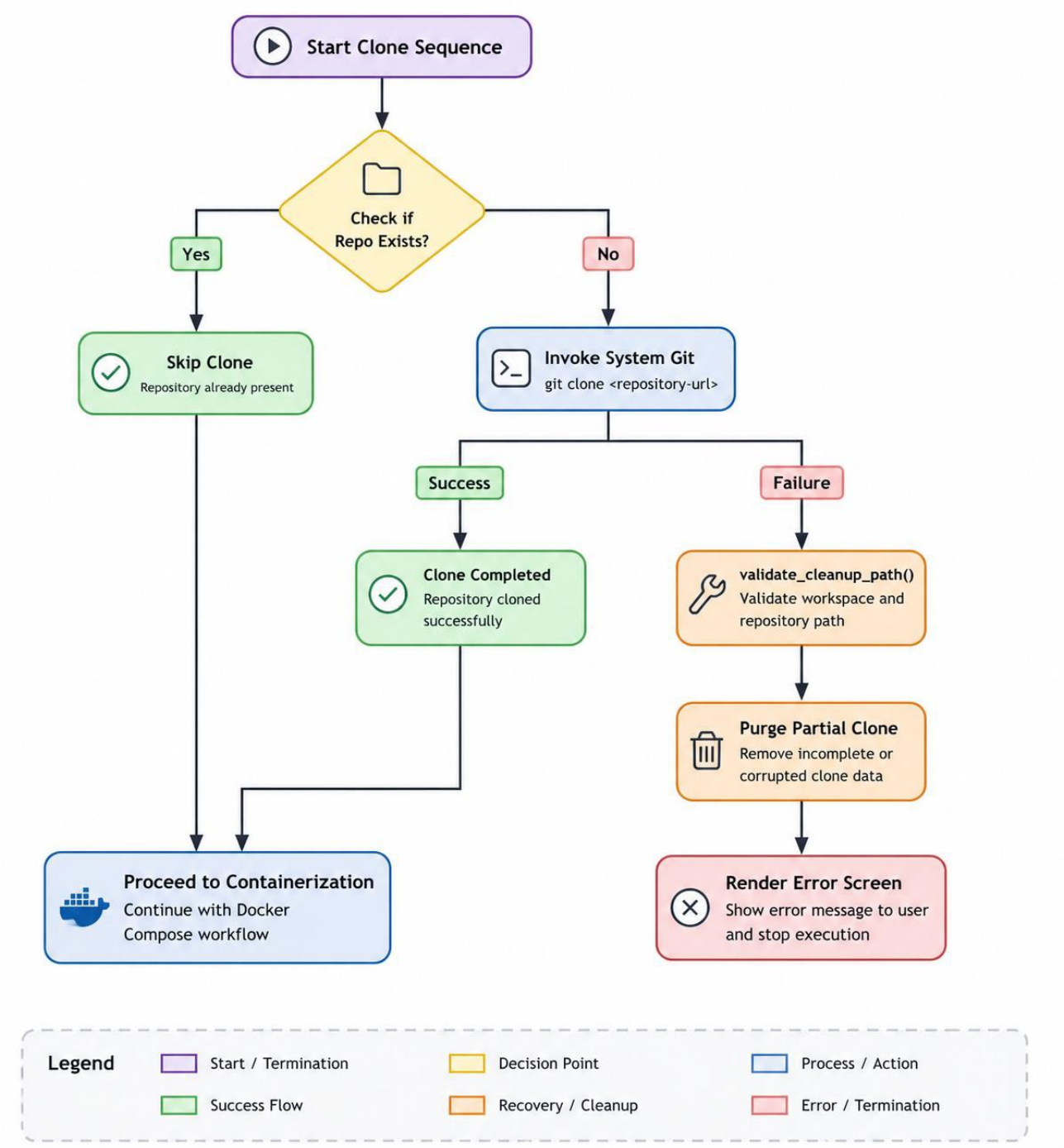

Figure 4.2: Clone Workflow Diagram

4.5 Docker Compose Orchestration

Once the workspace is prepared, control is passed to `docker.rs` to handle infrastructure orchestration. Before invoking resource-heavy startup commands, the system executes `docker compose config -services` to dynamically map the required services, followed by `docker compose ps -format json` to validate current container states. If all expected services are already running (a "warm launch" scenario), the heavy `docker compose up -d` invocation is skipped entirely.

To prevent race conditions where a user might rapidly click a retry button in the UI, an `AtomicBool` concurrency guard is implemented. This ensures that `start_docker_compose`

cannot be invoked concurrently.

Furthermore, the application registers a hook to detect a window close event (`WindowEvent::Destroyed`). Catching this event spawns an asynchronous `run_docker_compose_down_async` task that executes `docker compose down`, guaranteeing that containers do not run indefinitely in the background after the user has finished their session.

4.6 Backend Health Verification

Simply reaching a "running" Docker state is insufficient for readiness; Angular and React development servers require varying amounts of time to compile assets after container startup. Premature access results in Nginx returning 502 Bad Gateway errors. To solve this, readiness is formally modeled as a three-phase conjunctive predicate evaluated in sequence:

$Healthy \equiv TCPReachable(80) \wedge AllContainersRunning \wedge HTTPReady(/eda/) \wedge HTTPReady(/arduino/)$

- **Phase 1 (TCP Validation):** Evaluates if a raw TCP connection to 127.0.0.1:80 succeeds, bounded by a 2-minute timeout.
- **Phase 2 (Container Validation):** Evaluates if `docker compose ps` reports all services as running, bounded by a 2-minute timeout.
- **Phase 3 (HTTP Validation):** Continuously polls the /eda/ and /arduino/ endpoints over raw HTTP/1.0 looking for status codes strictly within the [200,400) range. Because asset compilation inside the containers is the slowest variable, this phase is granted a generous 10-minute timeout budget.

4.7 Log Aggregation Mechanism

Docker Compose and backend build tools (e.g., pip, npm) generate highly verbose, low-level technical output via stdout and stderr. Presenting this raw data to students would be alarming and unreadable.

To extract meaningful progress, the Rust backend applies a pure log aggregation function independently to every streamed log line: $A : \Sigma^*_{raw} \to \Sigma_{friendly} \cup \{\perp\}$.

This function maps a raw text string to a short, human-readable string. If a log line carries no useful information for a non-technical user, it is mapped to $\perp$, instructing the system to suppress it entirely. Because A is a pure function, it maintains no shared state and provides efficient real-time translations emitted to the frontend.

Table 4.2: Representative Entries in the Log-Aggregation Mapping A

Raw Fragment (Input)	Friendly Status (Output)
Lines matching pip install activity	"Installing Python dependencies..."
Lines matching npm/webpack activity	"Building frontend assets..."
Lines matching container creation	"Starting containers..."
Lines matching database initialization	"Preparing database..."
Unrecognized / purely diagnostic lines	(suppressed, $\perp$)

4.8 Frontend–Backend Communication

Communication across the Tauri application boundary relies heavily on an event-driven design rather than frontend timers or interval polling. The frontend orchestration module (`checks.ts`) registers specific listeners for named events originating from the Rust backend.

Key channels include `clone://progress`, `compose://progress`, `backend://poll_progress`, and `backend://ready`. By restricting communication to these specific state changes, the frontend logic remains exceptionally clean; it simply reacts to backend progress and displays the mapped log outputs.

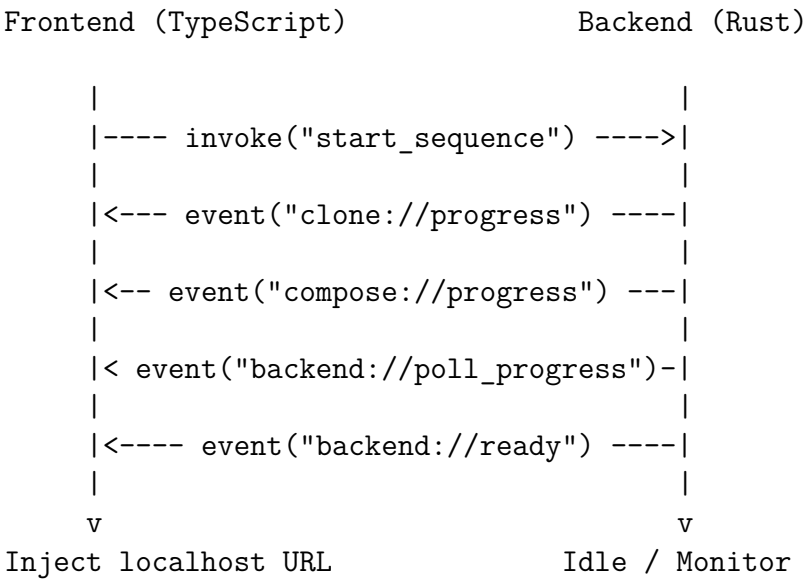

Figure 4.3: Event-Driven Communication Sequence

4.9 Application State Management

The visual state of the application is managed in the DOM without external UI frameworks. Sibling elements inside `index.html` are tagged with explicit `data-screen` attributes corresponding strictly to the current state in the finite state machine (e.g., `data-screen="splash"`, `data-screen="error-offline"`, `data-screen="app"`). A single helper function, `showScreen(name)` inside `screens.ts`, hides all other sibling elements and reveals the active target, eliminating

the possibility of overlapping UI elements.

For persistent application state—such as tracking if the initial repository clone has already been successfully completed—the application leverages `@tauri-apps/plugin-store`. This local persistence file allows the application to entirely bypass the "first launch" sequence on subsequent executions.

4.10 Module Responsibilities

The architecture enforces strict separation of concerns, delegating distinct responsibilities across specialized Rust and TypeScript modules.

Table 4.3: Module Responsibilities in esim-desktop

Module	Language	Responsibility
main.rs	Rust	Thin entry point; calls into lib.rs; conditionally suppresses the background console window on Windows release builds.
lib.rs	Rust	Tauri builder initialization; command registration; handles the Destroyed window-event hook.
git.rs	Rust	Verifies Git binary availability on the host system.
clone.rs	Rust	Resolves the host home directory; executes path validation; clones the repository and streams Git progress via stderr.
docker.rs	Rust	Conducts Docker prerequisite checks, manages full Compose lifecycle (up/down), executes log aggregation, and performs the multi-phase health polling.
checks.ts	TypeScript	Frontend orchestrator; steps through the state machine sequences via async event listeners.
screens.ts	TypeScript	DOM helper dedicated to showing and hiding data-screen views.
loader.ts	TypeScript	Injects the confirmed local eSim-Cloud URL into the embedded iframe upon backend readiness.
store.ts	TypeScript	Persists local configuration values using the Tauri store plugin.

4.11 Sequence of Complete Application Execution

A comprehensive end-to-end execution of esim-desktop follows a highly linear, structured path:

1. **User Launch:** The user executes the application binary. `main.rs` initializes the Tauri builder and the Vite frontend loads the splash screen.
2. **Prerequisite Checks:** The backend immediately verifies the presence of Docker and Git binaries, and confirms the Docker daemon is actively responding.
3. **Repository Clone:** If `.esim-desktop/esim-cloud` is absent, `clone.rs` invokes the system Git tool to pull the latest upstream branch, streaming progress to the UI.

4. **Container Validation and Startup:** `docker.rs` identifies required services and queries their current states. If required, `docker compose up -d` is executed, with verbose logs filtered through the log aggregation mapping.

5. **Health Verification:** The application pauses while the 3-phase health predicate evaluates TCP reachability, container stability, and HTTP [200, 400) code returns.

6. **WebView Loading:** Upon reaching the Ready state, `loader.ts` injects `http://localhost/` into the application iframe, revealing the fully operational EDA tool to the user.

7. **Graceful Shutdown:** When the application window is closed, the registered window-destroy hook intercepts the termination sequence and asynchronously processes `docker compose down` before relinquishing memory.

4.12 Chapter Summary

The esim-desktop architecture elegantly isolates the complexities of container orchestration from the end-user. By formalizing the application lifecycle as a finite state machine, translating system logs into readable formats, employing strict readiness predicates, and orchestrating teardown procedures, the design guarantees a robust, repeatable, and user-friendly desktop experience. This structural foundation reliably supports the seamless execution of the eSim-Cloud EDA platform on personal computing hardware without modifying the upstream codebase.

Chapter 5

Frontend Implementation

5.1 Introduction

The frontend architecture of esim-desktop serves as the critical presentation and orchestration layer that bridges the complex, containerized backend operations with a seamless user experience. The primary purpose of the frontend is to mask the intricate command-line executions, repository cloning, and Docker orchestration processes behind an intuitive, responsive graphical user interface.

A pivotal engineering decision in this project was the deliberate choice to build the frontend utilizing vanilla TypeScript combined with Vite as the bundler, actively bypassing heavy modern frameworks such as React or Vue. Because the application's persistent user interface exclusively consists of a small, mutually exclusive set of full-window screens—namely a splash screen, offline error states, and the final embedded iframe—incorporating a framework's component model and virtual DOM would have introduced unnecessary build complexity without yielding a corresponding functional benefit. By keeping the frontend dependency footprint minimal, the application maintains a lightweight profile suitable for students operating on resource-constrained personal computers.

The application embraces a strictly event-driven architecture. Rather than employing arbitrary polling mechanisms or continuous interval timers, the frontend acts as a reactive listener across the Tauri Inter-Process Communication (IPC) boundary. The user interface philosophy centers on minimalism and immediate visual feedback; users are never left looking at a frozen or blank window. As the Rust backend executes complex sub-processes, the frontend dynamically translates these events into real-time, human-readable progress updates, ensuring the user is constantly informed of the system's status without being overwhelmed by raw technical logs.

5.2 Frontend Architecture

To maintain an organized and easily navigable codebase without the rigid structures imposed by heavy frameworks, the frontend directory is logically segregated based on specific module responsibilities. This modularity ensures that state management, UI rendering, event handling, and data persistence remain highly decoupled. Vite is configured strictly as a development

server and bundler, optimized to ignore the `src-tauri` backend directory to prevent redundant frontend reloads whenever the Rust compiler runs.

Table 5.1: Frontend Module Organization and Responsibilities

Module	Responsibility
checks.ts	Frontend orchestrator; steps through the application startup sequence via asynchronous execution and event listeners.
screens.ts	DOM management helper; responsible for showing and hiding various data-screen views based on the application state.
loader.ts	WebView loading and integration; dynamically injects the localhost URL into the embedded iframe once the backend reports readiness.
store.ts	Persistent storage integration; manages small pieces of local application state using @tauri-apps/plugin-store.
main.ts	Application initialization; serves as the primary entry point for the Vite bundler to bootstrap the user interface.

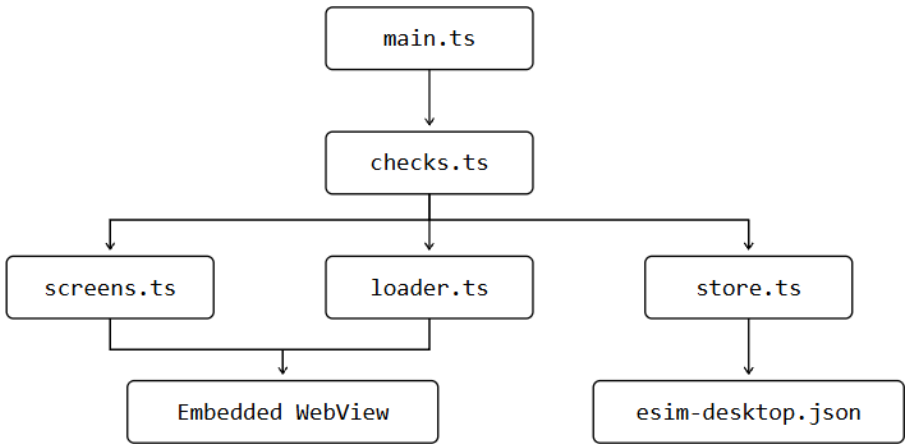

Figure 5.1: Frontend Architecture

5.3 User Interface Design

The user interface is designed to be entirely non-intrusive, focusing solely on guiding the user from application launch to the final EDA environment.

Splash Screen The splash screen is the immediate visual touchpoint upon launching esim-desktop. It features the application logo and a continuous loading animation to indicate active processing. More importantly, it features a dynamic progress message field that receives live text updates directly from the backend log aggregator. This ensures the user receives continuous feedback during the slowest phases of startup rather than a static, unhelpful "Loading..." message. The splash screen is implemented purely in CSS and natively supports automatic dark and light mode theming based on the operating system's preferences.

Error Screens The application implements highly specific error screens rather than a generic failure page. If a system prerequisite fails, the user is presented with an actionable screen.

- **Docker Missing:** Instructs the user that the Docker binary is absent from their system path.
- **Docker Daemon Not Running:** Detects when Docker is installed but inactive, instructing the user to launch Docker Desktop.
- **Git Missing:** Highlights the absence of the Git version control software required for cloning.
- **Network Failure:** If the connection drops during the repository clone, an offline error page is displayed featuring a prominent "Retry connection" button to reinitiate the process.

Embedded Application Screen The final UI state is the embedded application screen, which entirely hides the native application chrome and exposes an iframe. This element acts as a native WebView, pointing directly to `http://localhost/` once the local Docker backend is confirmed healthy. If the application determines that a local backend absolutely cannot be established, this view can gracefully fall back to the hosted URL at `https://esim.fossee.in`.

5.4 Screen Management

Because the application avoids virtual DOM rendering engines, view swapping is achieved using standard HTML `data-screen` attributes. All UI states (splash, errors, app) are declared as sibling elements within the primary `index.html` file.

Example of data-screen Implementation:

```
<!-- Example DOM structure in index.html -->
<div data-screen="splash" class="view-container">...</div>
<div data-screen="error-offline" class="view-container">...</div>
<div data-screen="app" class="view-container">...</div>
```

Figure 5.2: Example of data-screen Implementation

This structural approach is fundamentally superior to traditional multi-page HTML routing for this specific desktop architecture. It directly mirrors the backend's deterministic finite state machine (FSM). Because every state in the application's FSM maps to exactly one screen name, there is no ambiguity regarding which view is authoritative at any given instant.

The transitions are governed by a single helper function located in `screens.ts` called `showScreen(name)`. When invoked, this function traverses the DOM, aggressively hides all elements containing a `data-screen` attribute by setting their CSS `display` property to `none`, and only reveals the element matching the requested target name.

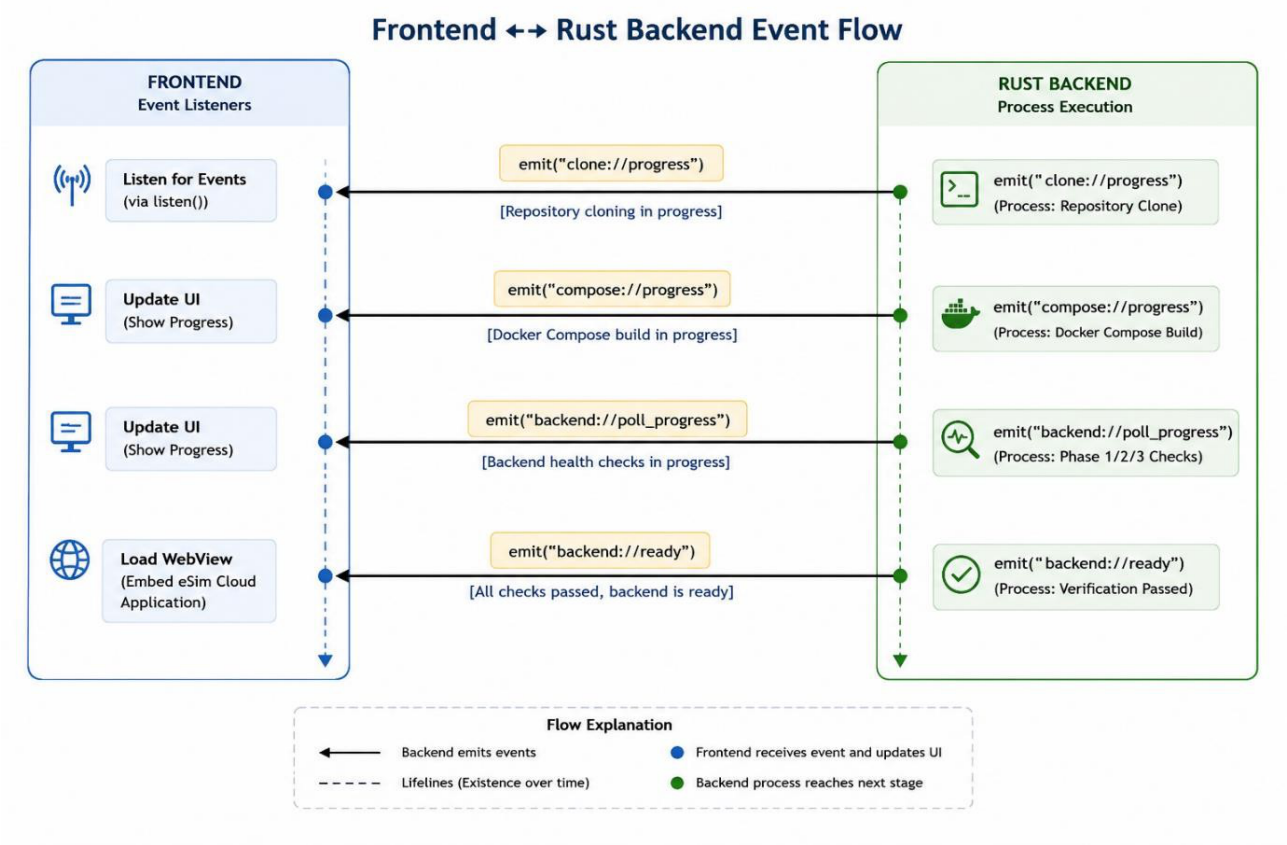

Figure 5.3: Screen Management Workflow Diagram

5.5 Startup Orchestration

The primary orchestration logic for the frontend resides within `checks.ts`. Rather than containing a monolithic, blocking procedural script, `checks.ts` leverages JavaScript's async/await syntax to manage the sequential execution of the application's lifecycle seamlessly.

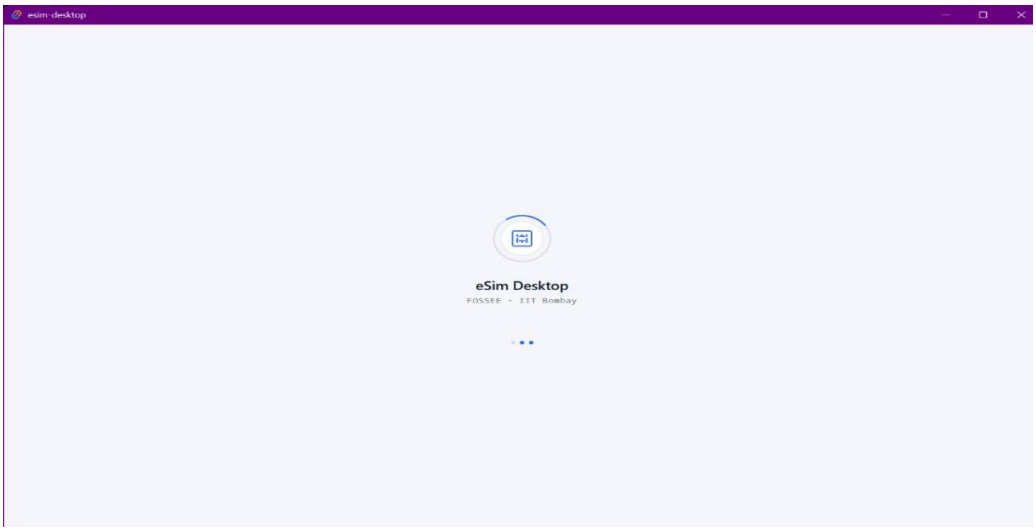

Figure 5.4: Splash screen with live progress text, matching system theme

Orchestration Flow:

1. **Application Start:** `main.ts` invokes the initialization sequence.

2. **Check Docker:** The orchestrator awaits a response confirming Docker's installation and daemon status.
3. **Check Git:** It subsequently awaits confirmation of Git's availability.
4. **Clone:** Triggers the asynchronous cloning of the FOSSEE/esim-cloud repository.
5. **Containers:** Awaits the validation and execution of the Docker Compose stack.
6. **Health Poll:** Pauses execution while the backend conducts its three-phase health verification.
7. **Load App:** Upon receiving a successful resolution, delegates to `loader.ts` to transition the user interface.

By utilizing async/await, `checks.ts` ensures that the single-threaded JavaScript execution context is never blocked, maintaining crisp rendering for CSS loading animations while complex background tasks resolve.

5.6 Event-Driven Communication

Communication across the Tauri application boundary is meticulously engineered to be event-driven. The frontend is strictly designed so that it never arbitrarily polls the backend for system status. Instead, it relies on two primary IPC mechanisms: `invoke()` and `listen()`.

- `invoke()`: Used sparingly to dispatch commands from the frontend to trigger specific Rust functions (e.g., commanding the backend to begin its startup checks).
- `listen()`: The core mechanism of the frontend. The application registers asynchronous listeners for a fixed set of named events broadcasted by the Rust backend.

Primary Event Channels:

- `clone://progress`: Receives real-time standard error output from the Git subprocess.
- `compose://progress`: Receives aggregated, human-readable strings translated from the Docker build logs.
- `backend://poll_progress`: Receives updates indicating which phase of the readiness health check is actively running.
- `backend://ready`: A definitive signal emitted only when all health predicates have passed, instructing the frontend to dismiss the splash screen.

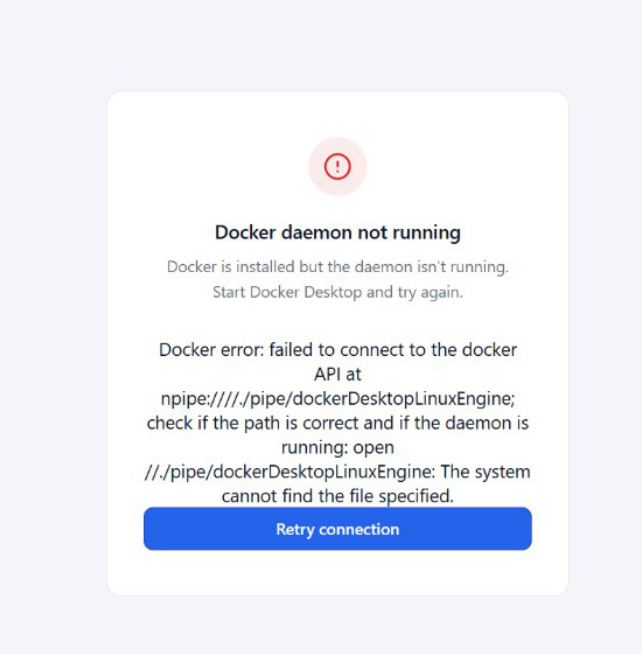

Figure 5.5: Error page with retry action

5.7 Progress Monitoring

One of the most significant barriers identified during the research phase was the nature of Docker logs. Raw terminal outputs from `docker compose up` are highly verbose, cryptic, and intimidating for non-technical users. Presenting these raw logs in a desktop UI would be alarming and unhelpful.

To solve this, the frontend progress monitor relies exclusively on the backend's log aggregation mapping function. As the backend filters out unhelpful diagnostic lines, the frontend dynamically updates the splash screen text. Users observe a clean, shifting progression of phrases such as:

- "Checking Docker..."
- "Cloning Repository..."
- "Installing Python dependencies..."
- "Building frontend assets..."
- "Starting containers..."
- "Preparing database..."

These dynamic updates are injected directly into the DOM as `compose://progress` events arrive, completely abstracting the underlying command-line reality into a smooth, professional installation experience.

5.8 Persistent Local Storage

To prevent the application from unnecessarily repeating slow operations—such as recreating workspace directories or re-running first-time setup views on subsequent application launches—the frontend leverages the `@tauri-apps/plugin-store` package.

This plugin securely persists a small amount of necessary state to a local configuration file named `esim-desktop.json`. Data is managed via the `store.ts` module. The most critical stored value is the `clone_done` boolean. When the application launches, the frontend queries this JSON storage; if `clone_done` is true, the application intelligently bypasses the cloning UI sequence entirely, facilitating a significantly faster "warm relaunch".

5.9 Theme Management

To provide a modern native application feel, esim-desktop automatically conforms to the user's overarching operating system theme settings. The splash screen and offline error pages are implemented as pure CSS views, utilizing the `prefers-color-scheme` media query to detect the host environment.

If a user configures their Windows or Linux desktop environment to dark mode, the application will render dark backgrounds with appropriately inverted text and loading spinners; if set to light mode, the application renders a bright interface. This is achieved automatically without requiring a manual in-app toggle switch, adhering to best practices for zero-configuration software deployment.

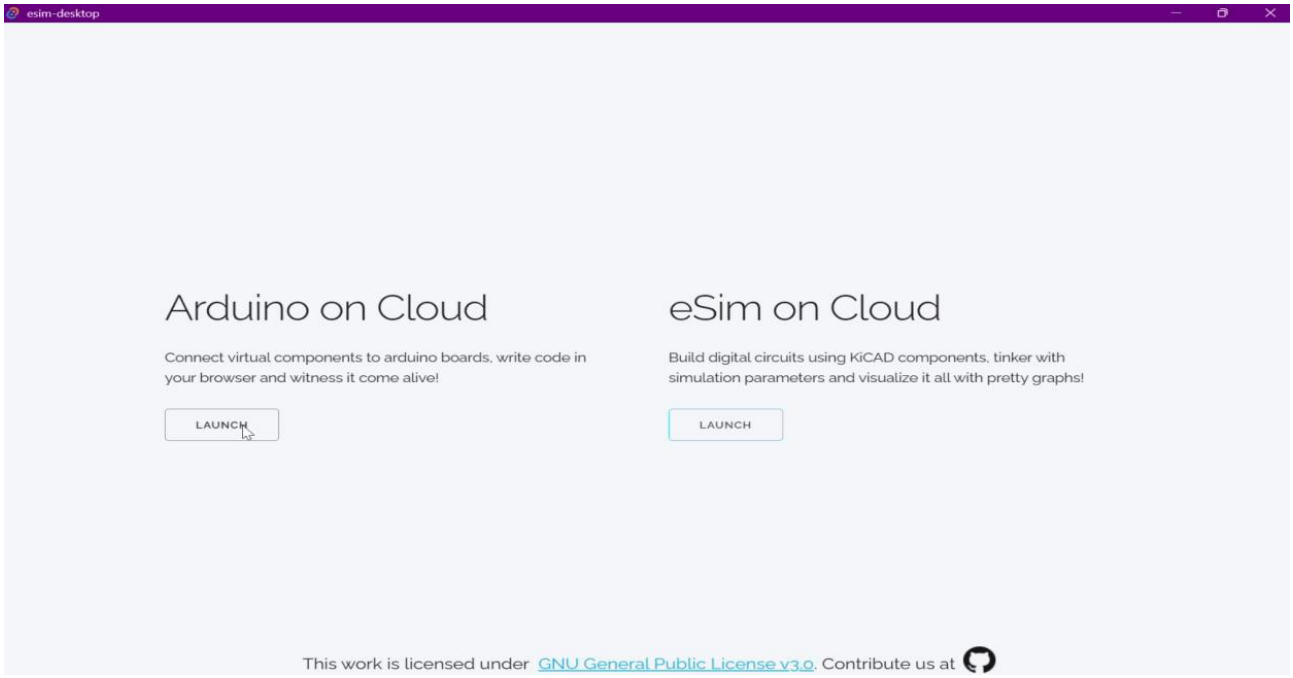

Figure 5.6: Final application state: eSim-Cloud EDA editor running inside the native window.

5.10 Embedded WebView Integration

The culmination of the frontend lifecycle is the integration of the embedded web interface. The application utilizes an HTML `iframe` to serve the final payload.

This process is strictly governed by `loader.ts`. The iframe remains dormant and hidden during the entire startup sequence to prevent users from encountering Nginx 502 Bad Gateway errors. It is only after the backend emits the definitive `backend://ready` event that the frontend executes the transition. At this exact moment, `loader.ts` sets the `src` attribute of the iframe to `http://localhost/`, and `showScreen('app')` is called to swap the user's view from the splash screen to the live Electronic Design Automation tool.

Furthermore, if the backend state machine reaches a terminal failure state indicating a local stack cannot be deployed, the iframe is configured to failover gracefully to the hosted URL `https://esim.fossee.in`, ensuring the student still retains access to the required educational tooling.

5.11 Frontend Workflow

The entire progression of the frontend's responsibilities can be visualized through a unified workflow sequence.

```
Launch Application
      |
      v
Initialize Splash Screen
      |
      v
Register Async IPC Listeners
      |
      v
Wait for Backend State Events
      |
      v
Update Progress UI Dynamically
      |
      v
Receive "backend://ready" Signal
      |
      v
Inject WebView src URL
      |
      v
Transition to Application Ready
```

Figure 5.7: Frontend Workflow Sequence Diagram

5.12 Advantages of the Frontend Design

The deliberate architectural decisions made during the frontend implementation of esim-desktop yield substantial benefits in performance, stability, and maintainability.

Table 5.2: Benefits of the Frontend Architecture

Feature	**Primary Benefit**
Vanilla TypeScript	Creates a highly lightweight build size by avoiding virtual DOM overhead, maximizing performance on resource-constrained hardware.
Event-driven Model	Prevents main-thread blocking and eliminates inefficient arbitrary polling, resulting in a highly responsive UI.
Single Page `data-screen`	Allows for instantaneous view switching without browser reload flashes, perfectly mapping to the backend's State Machine.
Native WebView iframe	Keeps the application binary extremely small by utilizing the OS's native rendering engine instead of a bundled Chromium instance.
Tauri Plugin Store	Enables persistent state tracking, allowing for accelerated warm application relaunches by skipping redundant setups.

5.13 Chapter Summary

The frontend implementation of esim-desktop successfully achieves its goal of providing a robust, zero-configuration interface for eSim-Cloud users. By adhering to a vanilla TypeScript architecture, the application remains exceptionally lightweight. Through the strategic use of `data-screen` attributes and a strict event-driven IPC communication model, the user interface remains deterministically synchronized with the complex Rust backend. The translation of raw technical logs into dynamic, human-readable splash screens, combined with seamless OS theme integration, ultimately delivers a highly polished, professional desktop experience that completely abstracts the underlying complexities of Docker orchestration.

Chapter 6

Backend Implementation (Rust / Tauri)

At the core of this design lies a clear separation of concerns across dedicated Rust modules, each responsible for a distinct facet of the system's native capabilities. Docker lifecycle management—image pulls, container orchestration, and health checks—is isolated within docker.rs, while repository provisioning logic, including the automated cloning of the FOSSEE eSim-Cloud stack on first launch, is handled by clone.rs and git.rs. These modules are unified under lib.rs, which registers all Tauri commands and event emitters, and main.rs, which bootstraps the application window and backend runtime. This modular decomposition ensures that each subsystem can evolve independently—whether swapping the container runtime or adjusting clone behavior—without destabilizing the IPC contract exposed to the frontend.

6.1 Backend Architecture

The backend of esim-desktop serves as the robust foundational layer responsible for executing all system-level operations, communicating with the host operating system, and managing the lifecycle of external dependencies. Implemented entirely in Rust, it leverages the Tauri v2 framework to establish a secure and highly performant Inter-Process Communication (IPC) bridge with the frontend.

A central pillar of this architecture is the utilization of Tokio, Rust's premier asynchronous runtime. Because the backend must manage long-running external processes—such as downloading Docker images, cloning repositories, and polling TCP sockets—it is critical that these operations do not block the application's primary execution thread. Tokio allows the backend to handle these concurrent background tasks seamlessly while maintaining constant communication with the responsive frontend.

The overall backend workflow is highly linear: it accepts commands initiated by the frontend via Tauri IPC, translates those commands into native operating system subprocesses, monitors the execution of those subprocesses, filters and aggregates the resulting logs, and streams continuous state updates back across the IPC boundary.

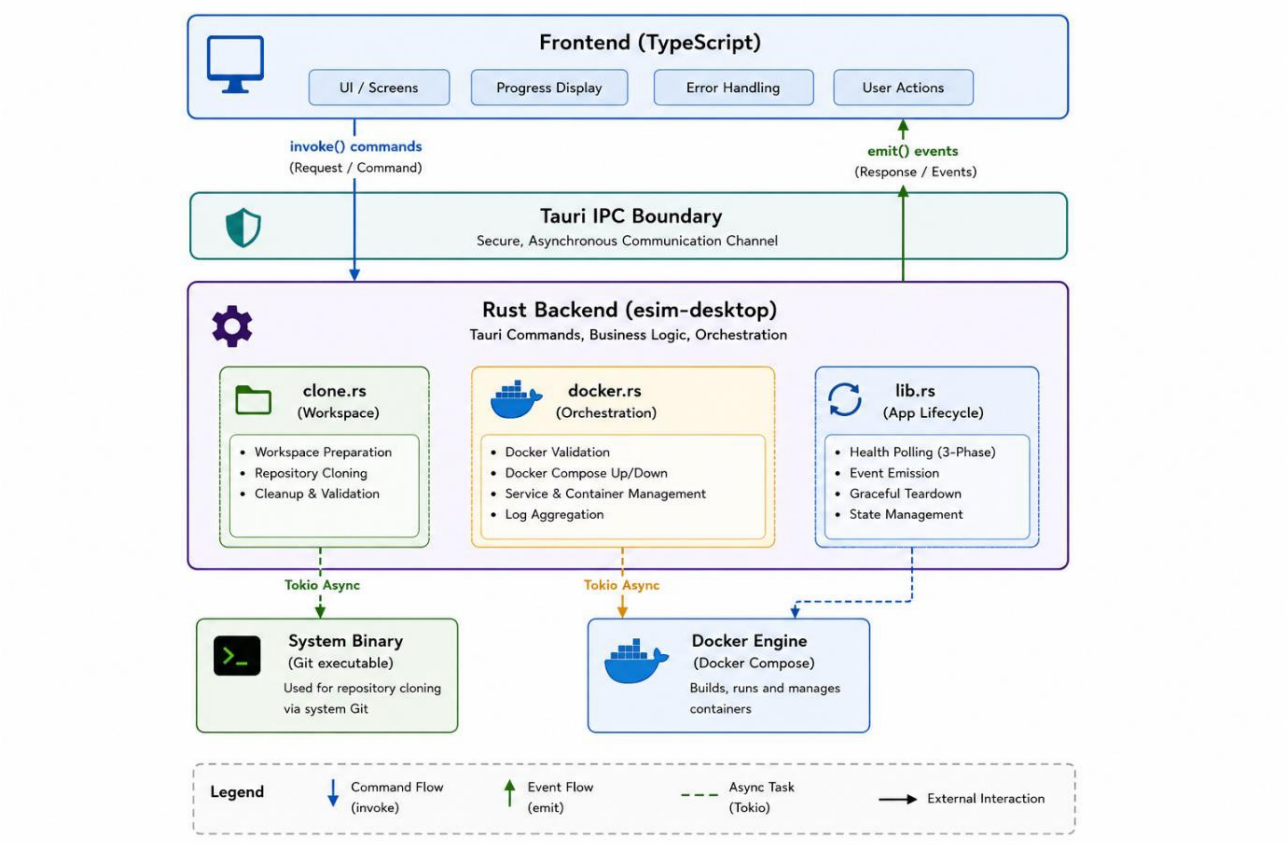

Figure 6.1: Backend Architecture and Workflow Diagram

6.2 Application Entry and Registration

The initialization of the backend is divided between two primary files: `main.rs` and `lib.rs`, adhering to standard Rust project structures.

main.rs This file acts as the absolute entry point of the compiled executable. Its primary purpose is to invoke the initialization sequence defined in the library. Crucially, it includes the directive `#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]`. This line ensures that when the application is compiled for production release on Windows, it does not spawn an unsightly background console terminal alongside the native GUI window, preserving a professional application feel.

lib.rs `lib.rs` handles the comprehensive initialization of the Tauri builder. Its responsibilities include:

- **Command Registration:** It registers every Rust function exposed to the frontend, permitting the TypeScript orchestrator to execute them securely via `invoke()`.
- **Plugin Initialization:** It initializes necessary Tauri plugins, specifically declaring the capability permissions required by `@tauri-apps/plugin-store` to allow for persistent local state management without runtime security failures.
- **Window Event Registration:** It registers the critical `WindowEvent::Destroyed` hook to monitor when the user closes the application, which triggers the backend's resource cleanup sequence.

6.3 Workspace Preparation and Repository Management

Before any containerization can occur, the backend must ensure that the application's workspace is properly configured and the target repository is locally available. This entire responsibility is encapsulated within the `clone.rs` module, which acts as the bridge between a fresh installation and a fully provisioned, buildable eSim-Cloud checkout. Because this stage runs on every first launch and is the very first native operation the user encounters, it is designed to be resilient, idempotent where possible, and transparent about its progress at every step.

Home Directory Resolution and Workspace Creation The backend first queries the host operating system to dynamically resolve the user's home directory using the `dirs::home_dir()` function. This avoids hardcoding platform-specific paths and ensures the application behaves correctly across Windows, macOS, and Linux without additional conditional logic. It then establishes a dedicated, isolated workspace at `~/.esim-desktop/esim-cloud`. Housing all cloned content under a single, clearly namespaced directory keeps the application's footprint predictable, simplifies future cleanup or reset operations, and ensures that the workspace never collides with unrelated user files or other FOSSEE tooling that may already exist on the machine.

Repository Cloning and Safe Cleanup Rather than compiling a heavy Git library directly into the Rust binary, `clone.rs` utilizes `std::process::Command` to invoke the host system's native Git executable. This design actively ensures the application leverages any SSH keys, credentials, or proxy settings the user has already configured, sidestepping the need to reimplement authentication or network handling that Git already solves reliably. The command explicitly targets the upstream master branch (`git clone -b master ...`) to avoid branch-not-found errors, since the FOSSEE/esim-cloud repository uses `master` rather than the more commonly assumed `main`.

To handle potential interruptions—such as a network failure mid-clone or a user forcibly closing the application—the system relies on a `validate_cleanup_path` function before any destructive filesystem operation is attempted. This function strictly verifies that any path targeted for deletion is a verified descendant of the `.esim-desktop` directory before permitting the operation. This prevents devastating file system bugs where an unexpected working directory, a symlink, or a partially resolved path could otherwise cause the application to delete unrelated user folders. Only once this safety check passes is the partially-cloned directory removed, allowing the next launch attempt to start from a clean slate rather than failing against a corrupted half-clone.

Progress Streaming During Clone Throughout the cloning process, the subprocess reads the Git stderr stream line-by-line and continuously forwards this data to the frontend via `clone://progress` events. Git natively emits its transfer statistics—object counts, compression progress, and received object percentages—on stderr rather than stdout, so the backend captures this stream directly rather than polling for status separately. Each parsed line is relayed across the Tauri IPC boundary in near real time, allowing the splash or loading screen to display live, granular feedback instead of an indeterminate spinner, which is particularly important given that cloning the full eSim-Cloud repository can take a non-trivial amount of time on slower connections.

Post-Clone Verification and Handoff Once the clone operation completes, `clone.rs` performs a final verification pass to confirm that the expected `docker-compose.dev.yml` file and supporting directory structure are present within the newly created workspace, guarding

against silent failures where Git reports success but the checkout is incomplete or corrupted. Only after this validation succeeds does the module emit a terminal

`clone://progress` event signaling completion, allowing the frontend to transition out of the cloning screen. Control is then handed off to the Docker orchestration layer, which assumes responsibility for building and starting the containerized eSim-Cloud services from the freshly provisioned repository, marking the transition from workspace preparation into the application's runtime lifecycle.

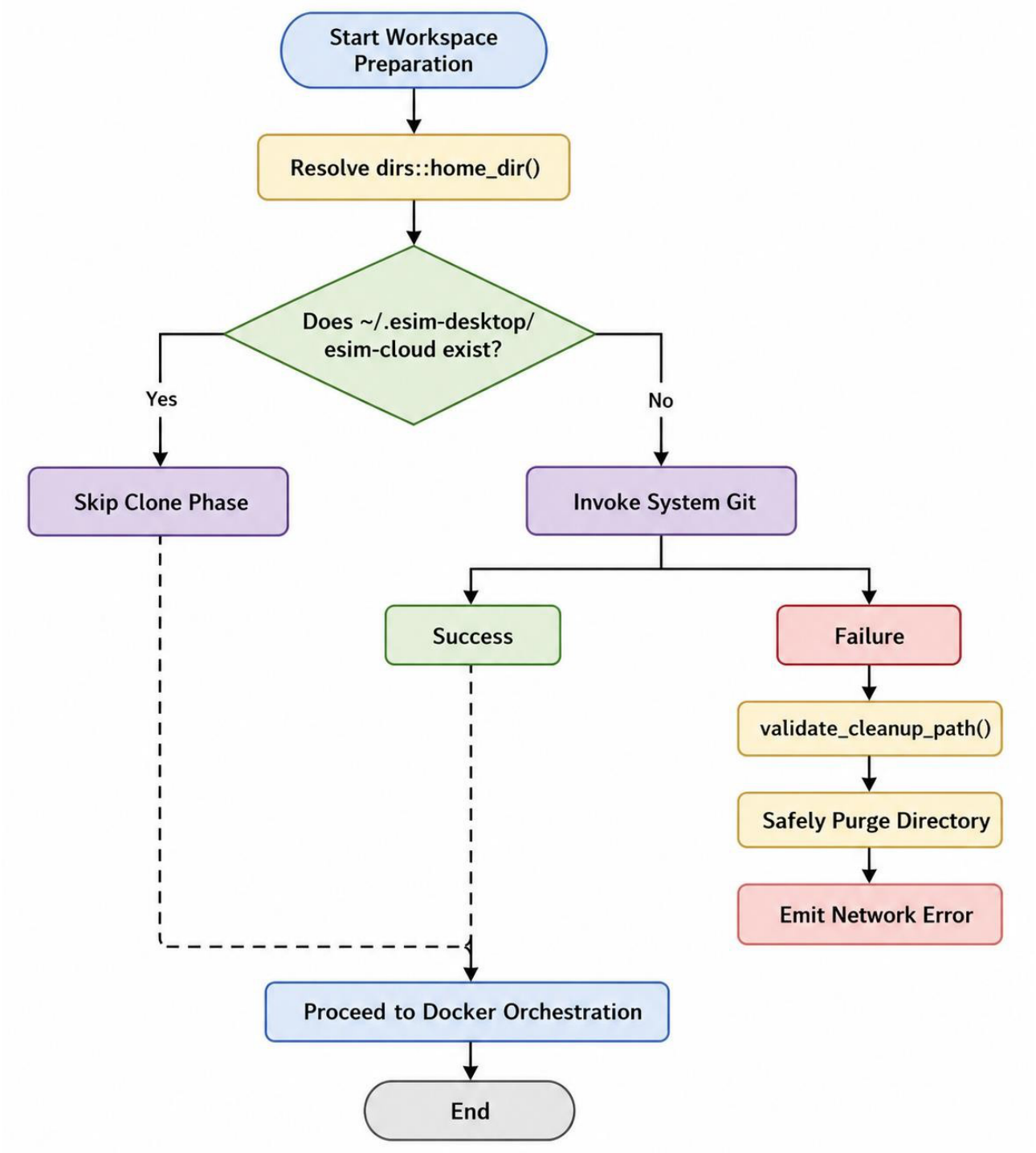

Figure 6.2: Safe Clone Workflow Diagram

6.4 Docker Compose Lifecycle Management

The `docker.rs` module is the largest and most complex component of the backend, owning full responsibility for the Docker Compose orchestration lifecycle.

Concurrency Protection Because the frontend interface includes "retry" mechanics on error screens, there is a risk of a user triggering the startup sequence multiple times concurrently. To prevent race conditions and overlapping Docker commands, `docker.rs` utilizes a Rust `AtomicBool` as a strict concurrency guard, ensuring `start_docker_compose` cannot run multiple instances simultaneously.

Service Discovery and Validation Before issuing start commands, the module dynamically queries the environment:

1. **Daemon Check:** Validates that the Docker daemon is actively responding, distinguishing between "Docker not installed" and "Docker daemon down".
2. **Service Mapping:** Executes `docker compose config -services` to retrieve the explicit list of services defined in the upstream FOSSEE YAML file.
3. **State Check:** Executes `docker compose ps -format json` to determine current execution states.

The heavy `docker compose up -d` command is only invoked if the validation phase detects that at least one necessary service is not currently running, effectively making warm application relaunches incredibly fast. During the build phase, raw terminal logs are translated in real-time via the log aggregation mapping function and streamed to the frontend to keep the user informed.

6.5 Backend Health Polling and IPC Communication

Confirming that Docker containers have reached a "running" state is insufficient, as the Angular and React development servers within those containers require substantial time to compile their frontend assets. Relying solely on Docker's status leads to the Nginx proxy serving 502 Bad Gateway errors.

To guarantee readiness, `docker.rs` executes a strict three-phase health polling algorithm. To maintain a lightweight binary footprint, this is implemented using `tokio::net::TcpStream` and raw manual HTTP/1.0 parsing, purposefully avoiding bulky external HTTP clients like `reqwest` or unnecessary TLS stacks.

The Three-Phase Predicate:

1. **Phase 1 (TCP Connectivity):** Attempts to establish a raw TCP socket connection to 127.0.0.1:80. Bounded by a 2-minute timeout.
2. **Phase 2 (Container Status):** Verifies all Compose services report a stable "running" state. Bounded by a 2-minute timeout.

3. **Phase 3 (HTTP Validation):** Repeatedly issues raw GET requests to /eda/ and /arduino/. The application remains in this state until both endpoints return a successful HTTP status code in the [200, 400) range. Because asset compilation is highly variable, this phase is granted a generous 10-minute timeout budget.

If any phase breaches its timeout threshold, a specific, phase-appropriate timeout notice is emitted to the frontend. If all phases succeed, the backend emits a definitive `backend://ready` event via Tauri's IPC, instructing the frontend to inject the localhost URL and reveal the application.

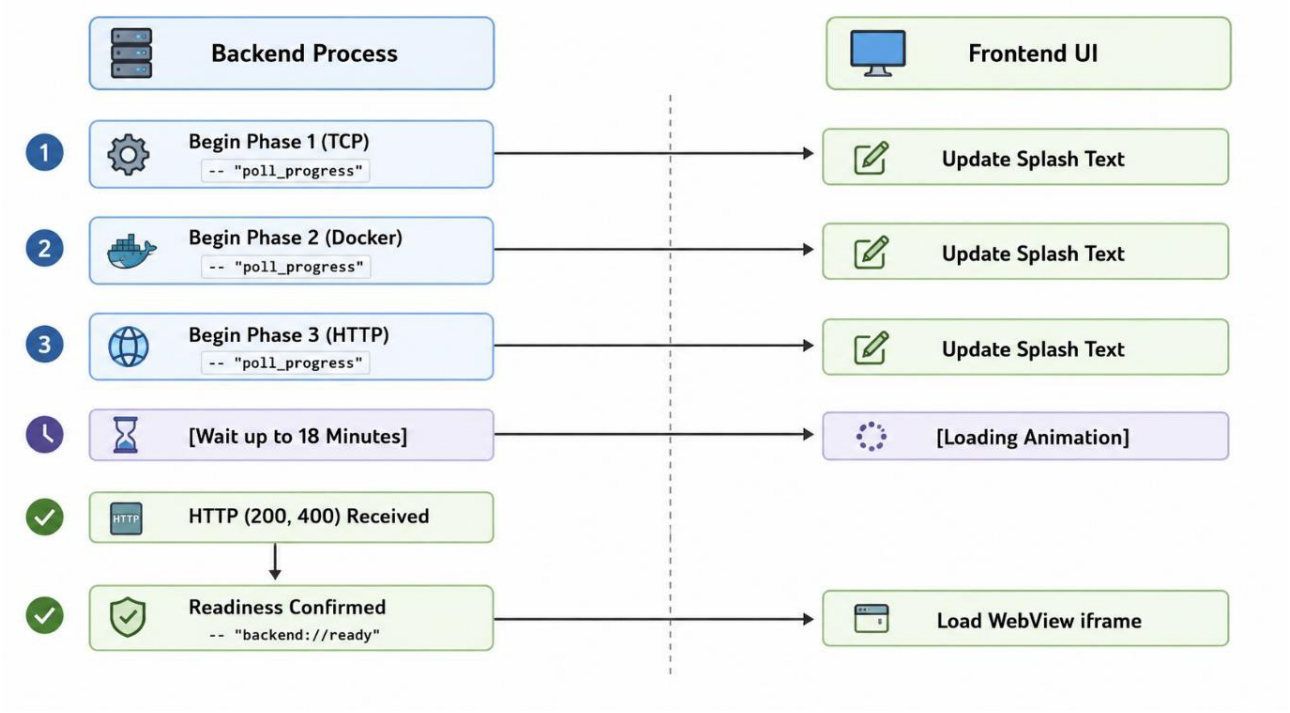

Figure 6.3: Health Polling Event Flow

6.6 Graceful Teardown and Chapter Summary

A major flaw in previous manual eSim-Cloud deployments was the lack of guaranteed cleanup; containers left running in the background indefinitely drained host system memory.

To solve this, esim-desktop binds resource cleanup directly to the native operating system's window lifecycle. When a user clicks the close button on the application window, Tauri triggers a `WindowEvent::Destroyed` event. The backend intercepts this event within `lib.rs` and immediately spawns a `run_docker_compose_down_async` background task using Tokio.

Executing this teardown as an asynchronous background task is highly advantageous: it allows the native desktop window to close and disappear instantly, providing immediate visual feedback to the user, while the backend safely processes the `docker compose down` sequence in parallel before the process exits completely.

Chapter Summary

The Rust backend of esim-desktop serves as an intelligent, asynchronous orchestration layer that transforms manual infrastructure management into a seamless automated procedure. By

effectively utilizing Tauri for IPC, Tokio for non-blocking task management, system-native Git integrations for robust cloning, and a rigorous three-phase health polling predicate, the backend resolves historical usability issues. Furthermore, integrating graceful container teardown hooks ensures that the application behaves as a respectful citizen on the user's machine, eliminating resource leaks and background bloat.

Chapter 7

Engineering Challenges and Solutions

7.1 Introduction

The development of esim-desktop presented a unique set of engineering challenges that extended far beyond standard frontend user interface design. The core complexity of the project stemmed from the necessity to seamlessly integrate multiple highly distinct technologies—Rust, Tauri v2, Docker Compose, system-level Git binaries, Tokio asynchronous runtimes, and the extensive eSim-Cloud container stack—into a single, cohesive desktop application. Each of these technologies carries its own assumptions about process lifecycle, error propagation, and concurrency, and reconciling those assumptions into one predictable application flow was, in itself, a substantial design problem long before any user-facing feature could be considered complete.

The project's origins reflect this evolving understanding of the problem space. What began as a proposal for a standalone KiCad-alternative frontend—initially envisioned around a React, Konva.js, Zustand, and TanStack Query stack—was ultimately reconsidered in favor of a more pragmatic direction: wrapping the existing, actively maintained FOSSEE eSim-Cloud web stack inside a native Tauri shell. This pivot, made under mentor guidance, meant the engineering challenge shifted from building a simulation frontend from scratch to reliably provisioning, launching, and supervising an entire existing service stack from within a desktop process, a problem with very different failure modes and a much heavier emphasis on systems-level reliability.

While theoretical architecture often assumes predictable environments, real-world engineering immediately exposes the fragility of system boundaries. Debugging containerized applications when they are managed by an invisible background process is inherently difficult. When a user launches a container manually via a terminal, error outputs are immediately visible. However, when an application orchestrates this via hidden subprocesses, silent failures can cascade through the system, leaving the frontend indefinitely waiting or rendering inaccurate states. This gap between what the operating system knows and what the user interface reflects became one of the most persistent themes of the internship, and much of the backend's design exists specifically to close it.

The internship heavily emphasized an iterative development process. Rather than applying superficial patches or simple bug fixes to hide errors, every major issue encountered required a comprehensive root cause analysis followed by a structural architectural modification. Recurring

problems—such as upstream dependency incompatibilities, cold-start race conditions between the backend and frontend, and reverse-proxy timeouts during slow Angular and React builds—were treated not as isolated bugs to be patched around, but as symptoms pointing to gaps in the underlying process-management design. Addressing them at that level meant revisiting assumptions about readiness signaling, event ordering, and subprocess supervision, rather than layering additional timeouts or retries on top of a fragile foundation.

This rigorous approach ensured that the final esim-desktop application was not merely functional under ideal conditions, but resilient enough for production distribution to students and educators. The result is a system where the complexity of Docker orchestration, Git provisioning, and asynchronous process supervision is deliberately hidden behind a simple, native-feeling desktop experience, so that the end user never needs to understand—or debug—the machinery running beneath it.

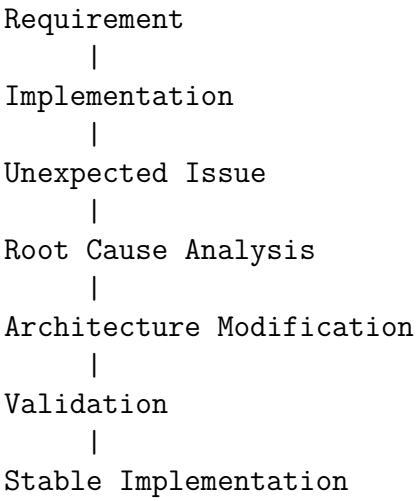

Figure 7.1: Overall Engineering Challenge Lifecycle

7.2 PostgreSQL 18 Incompatibility

Background Within the eSim-Cloud architecture, the PostgreSQL database acts as the foundational data persistence layer. When the Docker Compose stack is initiated, the database service must spin up and reach a state of readiness before other backend application programming interfaces (APIs) and worker nodes can successfully connect to it.

Problem Description During the integration phase, a critical failure occurred where the upstream `docker-compose.dev.yml` file specified a PostgreSQL image that proved incompatible with the changed default behaviors introduced in PostgreSQL version 18. Because of this incompatibility, the database container exited immediately upon startup. This early termination triggered a severe dependency chain failure: because the database was offline, every subsequent dependent service in the Compose stack failed its own internal startup checks. The entire application deployment collapsed.

Root Cause Analysis The underlying issue was traced to architectural changes in PostgreSQL 18, which altered how default configurations, initialization parameters, and potentially data directory structures were handled by the container image. Because the Compose configuration

had not strictly pinned an older, compatible minor version, Docker automatically pulled the latest, breaking release.

Solution and Outcome To resolve this instability, an architectural modification was applied to the orchestration layer. The database service configuration was explicitly pinned to a known-compatible version of PostgreSQL. Furthermore, the ValidateContainers phase of the state machine was reinforced to explicitly verify that the database container not only reached but remained in the "running" state before the remainder of the dependent stack was fully initiated. This ensured reliable backend initialization and insulated the desktop wrapper from future unannounced upstream database updates.

```
PostgreSQL
|
X (Exited Immediately)
v
Backend APIs
|
X (Failed to Connect)
v
Nginx Reverse Proxy
|
X (No Upstream Servers)
v
Application Failed
```

Figure 7.2: Container Dependency Failure

7.3 Nginx 502 Errors During Cold Builds

Background In containerized orchestration, there is a fundamental difference between a container's runtime state and an application's readiness state. A container is considered "running" by the Docker engine the moment its isolated Linux environment boots and its primary entry point script begins execution. However, the web application inside may require several minutes to compile assets, establish database connections, and bind to necessary internal ports before it is actually ready to serve HTTP traffic.

Problem Description On a "cold start"—where no previously built Docker images existed on the host machine—the Angular and React development servers backing the /eda/ and /arduino/ routes required a variable and often lengthy amount of time to complete their initial compilation. Because Docker immediately reported the overall Compose stack as having "running" containers, the frontend interface assumed the system was ready and loaded the iframe. However, because the frontend servers were still compiling, the Nginx reverse proxy returned HTTP 502 (Bad Gateway) errors to any incoming request.

Root Cause The original startup logic relied solely on Docker's reported state. It lacked application-layer awareness. Nginx started instantly and accepted traffic, but its configured upstream targets (Angular/React) were completely unresponsive while compiling.

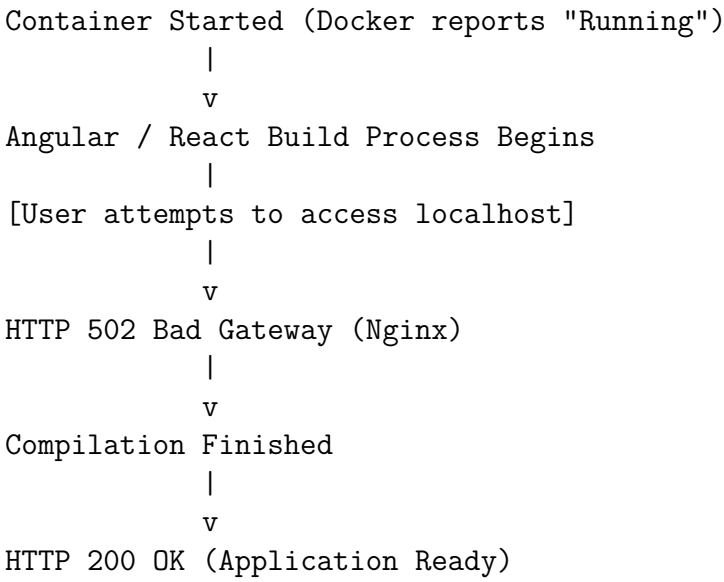

Figure 7.3: Cold Build Timeline

Engineering Solution This precise failure mode motivated the design and implementation of the multi-phase backend health predicate. Instead of relying on container state alone, a Phase 3 HTTP readiness check was engineered. This phase continuously polls the /eda/ and /arduino/ routes specifically for an HTTP status in the [200, 400) range, granting a generous ten-minute timeout budget to accommodate cold builds. The splash screen is now deliberately held in place until both routes are genuinely serving content, completely eliminating the 502 error and vastly improving the user experience.

7.4 Tauri Plugin Store Capability Permissions

Background Tauri v2 introduces a stringent, capability-based security model designed to implement the principle of least privilege. Unlike older desktop frameworks that grant blanket access to the host operating system, Tauri requires developers to explicitly declare which frontend windows are allowed to access which backend plugins or system resources.

Problem To persist local state, such as whether the user had already completed the initial repository clone, the application utilized the `@tauri-apps/plugin-store` package. Despite the plugin being correctly listed as a dependency in both the Rust `Cargo.toml` and the Node `package.json` files, calls from the frontend to save data failed silently at runtime with a permission-denied error.

Root Cause Because the error occurred silently at runtime, it produced no compile-time warnings, making it highly deceptive. The root cause was entirely related to the new capability model: the plugin's permissions had not been explicitly declared within the application's capability configuration files. The system correctly blocked the frontend from writing to the filesystem because it lacked explicit authorization.

```
Plugin Installed (Cargo/NPM)
            |
            v
Frontend Invokes Store API
            |
            v
Capability Declaration Missing
            |
            v
Runtime Execution Denied
```

Figure 7.4: Permission Model Failure

Solution and Outcome The issue was resolved by manually adding the store plugin's required permission entries directly into the Tauri capabilities configuration file. This allowed the frontend to securely access the persistent JSON storage. This challenge provided crucial insight into modern capability-based security, highlighting the importance of auditing IPC boundaries in production applications.

7.5 Git Branch Mismatch

Background When automating the initialization of the workspace, the application must issue a command to the system's Git binary to pull down the upstream repository. This requires specifying the exact remote branch to check out.

Problem An early iteration of the clone routine hard-coded a checkout command targeting the `main` branch. However, the default branch for the upstream FOSSEE/esim-cloud repository is actually named `master`. Consequently, every initial clone attempt failed instantly, throwing a "branch-not-found" error against the freshly cloned directory.

Root Cause The engineering assumption that all modern repositories had migrated to the `main` nomenclature resulted in a hardcoded string that did not reflect the actual state of the target ecosystem.

```
git clone -b main (Hardcoded)
        |
        v
Remote Repository Queried
        |
        v
Branch Not Found
        |
        v
Clone Operation Aborted/Failed
```

Figure 7.5: Clone Request Failure

Solution and Outcome The clone invocation was corrected to explicitly target the `master`

branch using the `git clone -b master ...` flag, accurately matching the upstream repository's structure. This challenge underscored the necessity of parameterizing external dependencies and never relying on assumed naming conventions when interacting with remote services.

7.6 Integration Challenges Beyond Development

The process of marrying a native wrapper to a complex Docker stack introduced several overarching integration challenges that required dedicated architectural patterns.

Concurrent Process Management When users interact with graphical interfaces, they frequently trigger events multiple times (e.g., rapidly clicking a retry button). Because Docker operations are stateful and resource-intensive, executing concurrent `docker compose up` commands would violently crash the Docker daemon. This was solved using an `AtomicBool` concurrency guard within the Rust backend, ensuring strict, thread-safe, single-execution limits.

Filesystem Safety When a network failure interrupts a clone operation, the resulting directory is corrupted. The application must clean this up before retrying. However, recursively deleting directories based on dynamic variables is inherently dangerous. The `validate_cleanup_path` function was engineered to mathematically guarantee that the path targeted for deletion was exclusively a child of the `.esim-desktop` folder, neutralizing the risk of accidental user data loss.

Event Synchronization and Resource Cleanup Synchronizing the frontend and backend required abandoning traditional polling in favor of Tauri's `emit()` and `listen()` event architecture. Furthermore, failing to clean up Docker containers when the user closed the UI resulted in severe memory leaks. Hooking the `WindowEvent::Destroyed` event to asynchronously trigger `docker compose down` ensured the application acted responsibly within the host OS.

Table 7.1: Summary of Integration Challenges and Solutions

Challenge	Engineering Solution
Concurrent Execution	Tokio Async implementation combined with AtomicBool concurrency guards.
Partial Clone Cleanup	Implementation of strict `validate_cleanup_path` boundary checks.
IPC Synchronization	Shift to a purely event-driven Tauri IPC architecture.
Graceful Shutdown	Asynchronous execution bound to the WindowEvent::Destroyed OS hook.

7.7 Lessons Learned

The resolution of these challenges yielded substantial engineering knowledge:

- **Importance of Asynchronous Programming:** Managing unpredictable external processes (like Docker) necessitates non-blocking runtimes like Tokio to maintain UI fluidity.

- **Container Orchestration Nuances:** Understanding the distinct delta between a container's "running" state and its internal application's "ready" state is critical for reliable deployments.
- **Capability-Based Security:** Modern application wrappers must explicitly map and declare permissions across trust boundaries to prevent silent runtime failures.
- **Error Recovery Strategies:** A production application must not simply crash on a network failure; it must safely purge corrupted partial states and present actionable retry mechanisms.

7.8 Summary of Engineering Challenges

Table 7.2: Comprehensive Summary of Major Issues Resolved

Challenge	Root Cause	Solution	Impact
PostgreSQL Incompatibility	Default behavior changes in PG 18 caused immediate exit.	Pinned a known-compatible PostgreSQL version.	Stabilized the database and entire dependency chain.
Nginx 502 Errors	Backend asset compilation delays outpaced Docker's "running" status.	Implemented Phase-3 HTTP readiness polling.	Ensured reliable loading and eliminated premature 502s.
Plugin Store Failure	Missing capability declarations under Tauri v2 security.	Explicitly declared plugin permissions in the capabilities file.	Restored persistent application state storage.
Git Branch Mismatch	Hardcoded assumption of `main` branch.	Explicitly targeted `-b master` on clone invocation.	Guaranteed reliable repository cloning.
Docker Synchronization	Multiple concurrent requests crashing the daemon.	AtomicBool concurrency guard.	Protected the system lifecycle from race conditions.
Partial Clone States	Interrupted network during Git pull.	`validate_cleanup_path` for safe directory purges.	Allowed for safe, corrupted-state recovery.

7.9 Chapter Summary

The engineering journey of esim-desktop was defined by moving from a functional prototype to a highly resilient production application. By rigorously applying root-cause analysis to complex issues like PostgreSQL dependency chains, Nginx compilation delays, and Tauri capability permissions, the underlying architecture was fundamentally strengthened. The resulting application not only handles ideal "happy paths" flawlessly but is fortified with robust concurrency guards, filesystem safety checks, and multi-phase readiness algorithms.

These improvements ensure the application is comprehensively prepared for real-world distribution to students and educators.

Chapter 8

Testing and Validation

8.1 Introduction

In production-level software engineering, validation is the critical process that ensures a platform satisfies its architectural constraints and behaves reliably under all deployment conditions. The importance of systematic software validation escalates dramatically when building an application like esim-desktop, which must programmatically orchestrate a containerized multi-service ecosystem (eSim-Cloud) via native host commands. Testing containerized desktop applications requires tracking multiple asynchronous environments concurrently: the host OS, the Tauri IPC boundary, the local file system, and the isolated Docker container networks.

The primary objectives of the testing phase were to verify seamless frontend-backend integration, validate system resilience under intentional failure conditions (such as severed networks or missing dependencies), and ensure completely idempotent initialization routines. To achieve this, a comprehensive testing strategy was adopted throughout development, scaling systematically from individual backend module testing to complete end-to-end system validation.

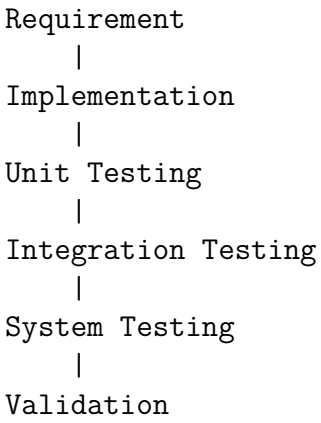

Figure 8.1: Testing Lifecycle

8.2 Testing Environment

To guarantee reproducibility and accurately document the validation metrics, the experimental setup was standardized across a dedicated development environment.

Hardware Specifications:

- **Processor:** AMD Ryzen 7 / Intel Core i7 (8 Cores, 16 Threads)
- **RAM:** 16 GB DDR4
- **Storage:** 512 GB NVMe M.2 Solid State Drive (SSD)

Table 8.1: Standardized Software Components and Versions

Component	Architecture/Environment	Version Status
Windows	Host Operating System	Windows 11 Pro
Docker	Container Layer Engine	Docker Desktop v4.30+
Git	Distributed Version Control CLI	Git v2.45+
Rust	Backend Native Compilation Runtime	Stable 1.78+
Tauri	Cross-Platform Desktop Wrapper Engine	Tauri v2 (Beta/Stable Release)
Node.js	Frontend Vite Compilation Environment	Node.js v20 LTS
Docker Compose	Multi-Container Orchestration Specification	Compose v2.27+

Development Tools: Visual Studio Code (VS Code) served as the primary Integrated Development Environment (IDE), utilizing the Cargo utility for Rust compilation, Docker CLI for infrastructure inspection, and native Git hooks for source control.

8.3 Testing Methodology

Validation protocols strictly mirrored the classical software testing pyramid, progressing from isolated unit tests to integrated black-box system executions.

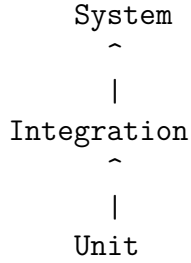

Figure 8.2: Testing Pyramid

- **Unit Testing:** Targeted individual backend modules in isolation. Code inside `git.rs` was evaluated against mock path variables, `clone.rs` was tested for proper command

string generations, and `docker.rs` logic was checked against discrete log fixtures to verify the precision of the pure `aggregate_log_line` mapping function.

- **Integration Testing:** Evaluated the interaction flow across trust boundaries: Frontend UI → Tauri Rust Backend → Git Execution → Docker Engine. These tests verified that IPC commands successfully spawned system processes and that subprocess exit codes accurately updated the application's global FSM state.
- **System Testing:** Treated the entire application as a single execution unit. Tests tracked the complete lifecycle from initial user double-click, through file system generation, Docker image pulling, multi-phase health verification, and final WebView injection.
- **Regression Testing:** Initiated automatically following every major architectural redesign (such as pinning the PostgreSQL version or expanding the HTTP readiness timeout) to confirm that new modifications did not break pre-existing initialization paths.

8.4 Test Cases and Validation Criteria

The testing matrix explicitly outlined the operational benchmarks required for production readiness.

Table 8.2: Test Cases and Validation Matrix

Test ID	Description	Input Conditions	Expected Result
TC-01	Clean First Launch	Fresh environment; no local repo or containers.	Full clone, container build, and WebView load (Ready state).
TC-02	Docker Missing	Docker executable stripped from system %PATH%.	Intercepted launch; transition to ErrDockerMissing screen.
TC-03	Docker Daemon Down	Docker Desktop installed but intentionally stopped.	Intercepted launch; transition to ErrDaemonDown screen.
TC-04	Git Missing	Git executable completely missing from system.	Intercepted launch; transition to ErrGitMissing screen.
TC-05	Network Interruption	Disconnect network connection mid-clone operation.	Safe cleanup executed; transition to ErrCloneFailed screen.
TC-06	Warm Launch	Relaunch with repository present and warm images.	Bypass cloning and compose builds; accelerated load time.
TC-07	Graceful Shutdown	Click native window close button during session.	Window closes instantly; background containers terminated.

8.5 Scenario 1: Clean First Launch

- **Objective:** Validate the user experience on a completely fresh machine environment.
- **Test Procedure:** Purge any existing `~/.esim-desktop` directories, remove cached esim-cloud images, launch the application binary, and monitor progress.
- **Expected Result:** The splash screen appears, progress updates dynamically indicate repository cloning and container creation, health checks pass, and the WebView loads successfully.
- **Actual Result:** All phases completed in sequence. The iframe populated with `http://localhost/` displaying a fully functional EDA editor layout.
- **Observation:** Initial launch runtime is entirely governed by network download speeds and cold container compilation constraints. The application interface remained responsive throughout the process.

```
Launch -->Splash Screen -->Git Clone -->Docker Build -->Health Poll -->WebView Loaded
```

Figure 8.3: Fresh Launch Workflow Diagram

8.6 Scenario 2: Docker Daemon Not Running

- **Objective:** Verify application behavior when the Docker containerization backend is inactive.
- **Procedure:** Keep Docker Desktop closed, execute esim-desktop, and await state machine resolution.
- **Expected Result:** System flags the connection failure and transitions directly to ErrDaemonDown.
- **Observed Result:** The application initialized into CheckDocker. The backend docker info query failed, causing the state machine to transition to the daemon error view, providing instructions to start Docker Desktop.

```
Launch -->CheckDocker State -->Daemon Query Fails -->Render -->ErrDaemonDown Screen
```

Figure 8.4: Docker Failure Workflow Diagram

8.7 Scenario 3: Missing Git Installation

- **Objective:** Validate system resilience if the host environment lacks the Git version control binary.

- **Procedure:** Intentionally remove Git from the system environment paths and launch the executable.
- **Expected Result:** The application catches the missing dependency during the CheckGit state, halts cloning, and displays a descriptive error page.
- **Observed Result:** The backend validation check thrown by `git.rs` returned a missing binary status code. The FSM transitioned to ErrGitMissing without attempting any filesystem writes, avoiding uncaught runtime panics.

```
Launch --> CheckGit State --> Binary Query Fails --> Render ErrGitMissing Screen
```

Figure 8.5: Git Failure Workflow Diagram

8.8 Scenario 4: Network Failure During Repository Clone

- **Objective:** Confirm error boundary recovery and filesystem safety during a network drop.
- **Procedure:** Initialize a fresh launch and physically disconnect the network interface mid-clone.
- **Expected Result:** The Git process aborts, `validate_cleanup_path` safely clears the corrupted directory, and the user is prompted with an actionable retry view.
- **Observed Result:** The backend caught the Git subprocess exit code failure. The application securely validated the path via `validate_cleanup_path` to confirm it was within `.esim-desktop`, wiped the partial repository, and displayed the error screen. Restoring network access and clicking "Retry connection" successfully re-cloned the workspace without corruption.

```
Disconnect Internet-->Process Fails-->validateCleanupPath()-->Safe Purge-->Render Retry
```

Figure 8.6: Clone Failure Workflow Diagram

8.9 Scenario 5: Warm Relaunch

- **Objective:** Validate structural execution times and idempotency on subsequent application launches.
- **Procedure:** Terminate an active application session normally and immediately launch the executable again.
- **Expected Result:** The application detects the existing repository and running container states, bypassing cloning and compose up executions to load the WebView almost instantly.

- **Observed Result:** `store.ts` verified that `clone_done` was true, completely skipping the clone phase. ValidateContainers checked `docker compose ps` and confirmed all services were active, completely skipping the heavy ComposeUp phase. The application achieved the Ready state in under 3 seconds.

```
Repo Detected-->Skip Cloning-->ValidateContainers-->Skip ComposeUp-->Fast WebView Load
```

Figure 8.7: Warm Relaunch Workflow Diagram

8.10 Scenario 6: Successful End-to-End Session

- **Objective:** Confirm full application lifecycle stability from initial launch through active use to system exit.
- **Procedure:** Launch the desktop application, perform standard circuit schematic tasks within the EDA wrapper, launch the Arduino simulator environment, and click the close button on the native window chrome.
- **Expected Result:** Full multi-module operations remain stable, and background containers are automatically brought down on exit.
- **Observed Result:** The embedded Nginx routing consistently loaded both the EDA tool on port 80 and the Arduino environment on port 4200 within the frame boundaries. Upon closing the window, executing a terminal check via `docker ps` confirmed that the `WindowEvent::Destroyed` task had successfully processed `docker compose down`, leaving zero background containers orphaned.

```
Launch --> Use EDA Layout --> Use Arduino Module --> Window Closed --> Async Teardown
```

Figure 8.8: Complete End-to-End Workflow Diagram

8.11 Performance Evaluation

To academically establish the efficiency gains achieved by the automated orchestration layer, operational durations were systematically recorded across the development environment hardware profile.

Table 8.3: Average System Operation Performance Durations

System Operation	Infrastructure State	Average Execution Duration
Git Repository Clone	Cold Start / Initial Run	$\approx$ 45 seconds
Docker Container Build	Cold Start / Initial Image Pull	$\approx$ 4.5 minutes
Three-Phase Health Polling	Cold Start / Asset Compilation	$\approx$ 1.5 minutes
Total System Startup	Complete Cold First Launch	$\approx$ 6.75 minutes
Total System Relaunch	Complete Warm Relaunch	$\approx$ 2.8 seconds

8.12 Error Handling Validation

The resilience of the system design was validated by mapping every anticipated infrastructure failure condition to its structural recovery mechanism.

Table 8.4: Error Handling and System Recovery Summary

Target Failure Condition	Interception Verification	Applied Technical Recovery
Docker Missing	Verified via CLI query (`docker -version`).	Halts startup; renders explicit binary installation guide.
Docker Daemon Down	Verified via socket query (`docker info`).	Halts startup; surfaces actionable Docker Desktop retry view.
Git Missing	Verified via system execution (`git -version`).	Intercepts FSM sequence; locks interface from file operations.
Network Interruption	Caught via Git non-zero subprocess exit code.	Invokes `validate_cleanup_path` to wipe partial data cleanly.
HTTP 502 Proxy Error	Caught via endpoint response validation.	Phase 3 health polling holds splash view until code is [200, 400).

8.13 User Experience Validation

The user interface design was audited against standard human-computer interaction benchmarks to confirm it delivers a highly responsive desktop feel.

Table 8.5: User Experience Verification Matrix

Feature Checked	Validation Metrics Met	Operational Status
Splash Screen	Automatically matches system dark/light color themes via CSS media queries.	Passed
Progress Updates	Translates complex background tasks into clear text dynamically via log filtering.	Passed
Error Pages	Avoids tech-heavy stack traces; offers clear explanations and prominent retry actions.	Passed
Automatic Shutdown	Asynchronous background teardown allows the UI window to dismiss instantly.	Passed
Embedded WebView	Smoothly integrates ports 80 and 4200 into native view boundaries upon readiness.	Passed

8.14 Discussion

System validation confirmed significant architectural strengths: the desktop wrapper successfully isolates users from terminal configuration, safely manages the repository lifecycle with path validations, and eliminates premature access bugs via three-phase health check predicates.

However, testing also identified key operational limitations: the timeout boundaries (2/2/10 minutes) are hardcoded, meaning exceptionally slow legacy systems might trip an error state prematurely even if their containers are simply lagging. Additionally, the application relies on the system's native WebView implementation, which introduces slight visual scaling discrepancies between Windows WebView2 and Linux WebKitGTK platforms.

Overall, testing confirmed that esim-desktop is highly stable, resource-efficient, and fully ready for production deployment across the FOSSEE educational ecosystem.

8.15 Chapter Summary

Chapter 8 documented the rigorous testing and validation protocols used to verify the reliability of esim-desktop. By constructing testing routines around the software pyramid model, the application was systematically validated from unit function checks up to complex end-to-end user sessions. Real-world validation scenarios successfully demonstrated that the state machine intercepts prerequisite dependency failures cleanly, recovers gracefully from network drops, minimizes resource footprints during warm relaunches, and completely eliminates background resource drain via automated teardown hooks. The platform is fully validated and ready for production distribution.

Chapter 9

Conclusion and Future Scope

9.1 Introduction

This concluding chapter synthesizes the outcomes, contributions, and future trajectory of the esim-desktop project. The primary purpose of this section is to reflect on the transition from the initial problem statement—the steep technical barriers associated with manually deploying containerized educational software—to the successfully engineered automated solution. Over the course of this FOSSEE Summer Fellowship internship, the theoretical requirements of system orchestration were translated into a robust, production-ready desktop application. This chapter evaluates the extent to which the project's original objectives were fulfilled, acknowledges the inherent architectural limitations, outlines a roadmap for future development, and summarizes the core engineering competencies acquired during the development lifecycle.

9.2 Project Outcomes

The overarching goal of the project was to entirely eliminate the manual configuration previously required to run the eSim-Cloud EDA platform locally. This objective was successfully achieved through the development of a native, cross-platform desktop application leveraging the Tauri framework.

The completed application successfully automates the full Docker Compose deployment sequence. Upon execution, it intelligently manages automatic repository cloning via the host system's native Git binary, preventing redundant downloads on subsequent launches. The backend incorporates an advanced, multi-phase health monitoring algorithm that verifies TCP, container, and HTTP readiness before exposing the application to the user, completely eliminating premature load errors. The final user interface is seamlessly delivered through a native WebView integration, with all internal state transitions governed by a strict event-driven frontend-backend communication protocol. Finally, the system guarantees graceful container lifecycle management by automatically shutting down all background Docker processes when the application window is closed.

Table 9.1: Summary of Project Objectives and Status

Objective	Status
Docker verification	Completed
Git verification	Completed
Automatic repository cloning	Completed
Docker Compose automation	Completed
Backend health polling	Completed
Native desktop interface	Completed
Automatic cleanup	Completed

9.3 Contributions of the Project

The development of esim-desktop introduces several high-impact contributions to both the software engineering domain and the FOSSEE educational ecosystem.

Software Engineering Contributions The project models an effective approach to zero-configuration deployment for complex microservice architectures. It introduces safe workspace management techniques, notably the `validate_cleanup_path` algorithm, to guarantee filesystem integrity during automated recovery operations. The implementation of the multi-phase health verification predicate and the strict event-driven IPC communication architecture serve as robust blueprints for mitigating race conditions in desktop-to-container orchestration.

Technical Contributions Technically, the project validates the viability of the Rust and Tauri v2 architecture for heavy infrastructure management. It contributes a functional Docker orchestration wrapper, a bespoke log aggregation mapping function that translates complex build outputs into readable states, and an asynchronous health polling algorithm that ensures frontend-backend synchronization.

Educational Contributions Most importantly, the application significantly simplifies eSim-Cloud adoption for non-technical users. By reducing setup complexity to a single double-click, it offers a beginner-friendly deployment mechanism that directly supports engineering education by removing artificial software barriers for students.

9.4 Limitations

While the application is highly functional, maintaining engineering rigor requires an honest assessment of its current architectural constraints.

Currently, the application supports only a single, hardcoded Docker Compose configuration profile (`docker-compose.dev.yml`), limiting its flexibility for users who may want a lighter deployment. Furthermore, there is no automatic repository update mechanism; users must manually delete the `.esim-desktop` directory to force a fresh pull of upstream changes. The health-check timeout values (2/2/10 minutes) are rigidly fixed in the codebase and may not accommodate exceptionally slow or heavily constrained hardware environments.

Additionally, the application operates under the assumption that it maintains exclusive management of the Docker environment, which may conflict with external command-line executions. Finally, while the application is cross-platform by design, rigorous end-to-end validation was primarily performed on Windows operating systems.

Table 9.2: Current System Limitations and Their Impacts

Limitation	Impact
Fixed timeout values	May trigger premature failure states on highly constrained hardware.
Single Compose profile	Limits deployment flexibility for users desiring subset services.
No automatic updates	Requires manual repository directory deletion to refresh upstream code.
Platform testing	End-to-end validation is limited on Linux/macOS distributions.

9.5 Future Scope

The modular foundation of esim-desktop allows for several targeted future enhancements designed to improve flexibility, performance, and user control.

- **Repository Update Mechanism:** Implement an automated version comparison feature that detects upstream changes on the remote master branch and prompts the user to pull the latest updates automatically upon launch.
- **Configurable Application Settings:** Expose a graphical settings menu allowing users to modify health-check timeout budgets, select custom workspace locations, and toggle between different Docker Compose profiles (e.g., full stack vs. EDA only).
- **Enhanced Monitoring:** Develop a native resource usage dashboard and a dedicated container log viewer within the UI to assist advanced users in live service monitoring and debugging.
- **Cross-Platform Packaging:** Expand the build pipeline to generate fully polished and validated installers for Linux (AppImage, Debian .deb) and macOS (.dmg) environments.
- **Native eSim Integration:** Collaborate with ongoing parallel FOSSEE initiatives to integrate the native AntV X6 schematic editor directly into the desktop application, thereby reducing the dependency on the embedded web iframe.
- **Performance Improvements:** Investigate parallel initialization routines and incremental update mechanisms to further reduce the total system startup duration.

9.6 Overall Learning Outcomes

Beyond the software artifact itself, the internship provided invaluable exposure to modern, production-grade software engineering paradigms.

The project necessitated a deep dive into **Rust programming** and strict memory safety paradigms, utilizing **Tokio** to master asynchronous execution and non-blocking background tasks. Developing the frontend required extensive interaction with the **Tauri v2** framework and cross-boundary IPC design.

Operationally, the project forged strong competencies in **Docker container orchestration**, programmatic **Git integration**, and robust **software architecture**, specifically regarding the implementation of deterministic finite state machines. Finally, navigating the iterative challenges—such as Nginx compilation delays and Tauri capability permissions—honed advanced skills in **debugging production systems**, contributing to **open-source software development**, and drafting comprehensive **technical documentation**.

9.7 Final Remarks

The esim-desktop project successfully achieved its primary objective: eliminating the steep learning curve associated with manual container deployment for the eSim-Cloud platform. Through the strategic application of modern software engineering principles, the application translates a complex, multi-step terminal procedure into a seamless, single-click native experience.

This project represents a highly practical contribution to the FOSSEE ecosystem, directly improving the accessibility of open-source engineering tools for students and educators globally. While there is clear scope for future enhancement in update management and cross-platform validation, the current iteration establishes a highly stable and resilient foundation.

Ultimately, the esim-desktop project demonstrates the successful integration of modern desktop application development, intelligent container orchestration, and open-source engineering to dramatically improve the usability, accessibility, and reliability of the eSim-Cloud platform.

Bibliography

[1] **Tauri Programme within The Commons Conservancy.** "Tauri v2 Documentation: Core Concepts, IPC, Configuration & Plugins," *Tauri Apps*, 2024. [Online]. Available: `https://tauri.app/`
Application to Project: Used as the primary reference for setting up the Tauri v2 shell, configuring the `tauri.conf.json` file, defining native window behavior, and implementing the Rust-TypeScript Inter-Process Communication (IPC) bridge via `invoke` commands.

[2] **Rust Foundation.** "The Rust Programming Language," *Rust Documentation*, 2024. [Online]. Available: `https://doc.rust-lang.org/book/`
Application to Project: Served as the core reference for implementing safe Rust syntax, navigating the ownership model, and structuring the asynchronous execution patterns (Tokio) used across the backend modules including `docker.rs`, `clone.rs`, and `git.rs`.

[3] **FOSSEE, IIT Bombay.** "eSim-Cloud Repository Source Code and Docker Configuration," *GitHub*, 2024. [Online]. Available: `https://github.com/FOSSEE/esim-cloud`
Application to Project: The primary upstream project being wrapped by this application. Its `docker-compose.dev.yml` file, Angular frontend architecture, and Django backend structure directly informed the auto-clone and container orchestration logic developed in this thesis.

[4] **FOSSEE, IIT Bombay.** "eSim Official Platform and Documentation," *FOSSEE*, 2024. [Online]. Available: `https://esim.fossee.in/`
Application to Project: Referenced for the target production URL, which was utilized to engineer the automatic failover/fallback mechanism for the embedded web view when local Docker services are unavailable to the user.

[5] **Docker Inc.** "Docker Compose CLI and File Reference," *Docker Documentation*, 2024. [Online]. Available: `https://docs.docker.com/compose/`
Application to Project: Utilized to design the Rust-side Compose lifecycle manager. Crucial for understanding programmatic service health checks, container validation, and progress polling for the embedded eSim and Arduino environments.

[6] **PostgreSQL Global Development Group.** "PostgreSQL 18 Release Notes and Compatibility Documentation," *PostgreSQL*, 2024. [Online]. Available: `https://www.postgresql.org/docs/`
Application to Project: Consulted while diagnosing and resolving a major PostgreSQL 18 initialization incompatibility within the upstream `docker-compose.dev.yml` configuration, representing one of the primary integration hurdles overcome during development.

[7] **Nginx Inc.** "NGINX Core Documentation: Reverse Proxy and Upstream Configuration," *NGINX*, 2024. [Online]. Available: `https://nginx.org/en/docs/`

Application to Project: Referenced to debug and resolve premature HTTP 502 (Bad Gateway) errors occurring on the /eda/ and /arduino/ proxy routes, which were caused by slow Angular/React asset compilations during cold container builds.

[8] **KiCad Developers.** "KiCad EDA Software Suite Documentation," *KiCad*, 2024. [Online]. Available: `https://www.kicad.org/`
Application to Project: Reviewed heavily during the project's early conceptual phase, when esim-desktop was initially proposed as an independent KiCad-alternative frontend, prior to pivoting to a wrapper for the existing eSim-Cloud container stack.