1.INTRODUCTION

Software development education is strongly influenced by tooling accessibility. Beginners frequently spend significant time configuring runtimes and compilers before writing their first executable program. This setup burden is a major barrier in classrooms, workshops, and self-learning environments. NimbusCode was designed to reduce this burden through a browser-native coding workflow.

NimbusCode treats the browser as the full execution platform. Instead of relying on remote servers for code execution, it uses client-side WebAssembly runtimes. This approach simplifies deployment and improves privacy because source code can remain on the user device.
In addition to execution capability, the project emphasizes interface familiarity. Students are already accustomed to the panel-based layout of modern IDEs. NimbusCode adapts this pattern with an explorer, editor tabs, language awareness, and output console, resulting in a low-friction learning experience.

1.1 INTRODUCTION TO THE PROJECT

NimbusCode is a static web application developed with React and TypeScript. Its architecture is deliberately backend-free. All primary operations, including file management, editing, and language runtime execution, are performed in the browser context.

The project uses the following technical pillars:

- **UI and state management:** React functional components and hooks.
- **Editor system:** Monaco Editor integration for syntax highlighting and editing.
- **Execution engine:** Runno runtime packages for WebAssembly language support.
- **Storage model:** IndexedDB for local workspace persistence.
- **Build and tooling:** Bun, Vite, TypeScript, ESLint.

NimbusCode currently supports seven languages and extension-based runtime mapping. The implementation includes specialized handling for C and C++ compilation, simple completion providers for selected languages, and configurable editor toggles.

1.2 STATEMENT OF THE PROBLEM

Educational institutions often face several recurring issues in programming labs:

- Heterogeneous operating systems and environment compatibility problems.
- High onboarding time due to package installations and configuration mismatch.
- System restrictions in lab devices that prevent runtime setup.
- Loss of instructional time resolving machine-specific development issues.
- Difficulty in ensuring consistent execution behavior for all students.

Cloud IDEs partially solve these problems but introduce infrastructure cost, account management, and internet dependency concerns. A fully local browser IDE can provide a middle path: low setup overhead with no backend execution infrastructure.

The problem statement for this project is: *Design and implement a browser-only IDE that supports multi-language execution and persistent workspace management without server-side code execution.*

1.3 SYSTEM SPECIFICATIONS

Minimum software requirements:

- Modern Chromium-, Firefox-, or WebKit-based browser with WebAssembly support.
- JavaScript enabled and compatibility with ES2022 features.
- Cross-origin isolated serving setup for runtime features requiring shared memory.

Development environment requirements:

- Bun runtime for package management and script execution.
- Node-compatible tooling ecosystem for TypeScript and ESLint.
- Vite development server with configured COOP/COEP headers.

Hardware recommendations for smooth usage:

- Dual-core or better CPU.
- Minimum 4 GB RAM; recommended 8 GB for heavy browser workloads.
- Stable internet connection for first-time runtime asset download.

Supported Languages in NimbusCode

Language	Extensions	Runtime Mapping
JavaScript	. js,. mjs, .cjs	quickjs
Python	.py	python
C	.c	clang
C++	.cpp, .cc, .cxx	clangpp
PHP	.php, .phtml	php-cgi
SQLite	.sql	SQLite
Ruby	. rb	ruby

2. LITERATURE SURVEY

Browser-based development environments have evolved significantly in the last decade. Early systems mostly offered text editing and remote execution, while modern systems support in-browser runtimes. The emergence of WebAssembly introduced portable high-performance execution in browser sandboxes, enabling interpreters and compilers to run client-side.

2.1 Survey Focus and Evaluation Criteria

The literature and tool survey for this project considered:

- Runtime model (server-side vs client-side execution).
- Setup complexity for beginner users.
- Offline capability and data persistence model.
- Security and isolation model.
- Pedagogical usefulness for introductory programming.

2.2 Comparative Discussion

Traditional desktop IDEs provide rich features and extensibility, but they require local installation and environment setup. In controlled lab settings, this often delays coursework.

Cloud IDEs centralize execution and simplify local setup but require backend resources, account onboarding, and network reliability. They may also introduce privacy concerns because source code is executed remotely.

Notebook platforms are effective for data-oriented workflows but less suitable for file-system-centric multi-language compilation use cases.

WebAssembly-native IDE approach aligns with local-first educational usage by keeping execution in-browser while maintaining an IDE-like interaction model.

High-Level Comparison of Development Environments

Aspect	Desktop IDE	Cloud IDE	NimbusCode Model
Setup effort	High (installation needed)	Medium (account and project init)	Low (open browser and start)
Execution location	Local machine	Remote server	Browser runtime (local)
Infra cost	Local maintenance	Continuous cloud cost	Static hosting only
Data location	Local filesystem	Provider-managed cloud storage	Browser IndexedDB
Primary constraint	Device setup complexity	Network and account dependency	Browser runtime limits

2.3 Key Takeaways for This Project

The survey indicates that a front-end-only browser IDE is a valid architectural choice for educational labs, quick experiments, and demonstrations. The most important design challenge is balancing simplicity and capability: adding enough IDE behavior to be useful while avoiding backend complexity.

2.4 References to Enabling Concepts

Core enabling concepts include:

- WebAssembly as a portable low-level execution target.
- WASI for standardized runtime interaction in sandboxed contexts.
- IndexedDB as structured client-side persistence.
- Component-based UI state management for interactive editor systems.

3. SYSTEM ANALYSIS

3.1 EXISTING SYSTEM

In many institutes, practical coding workflows depend on either pre-installed local compilers or web tools that perform remote execution. Existing workflows usually involve at least one of the following pain points:

- Frequent reinstall/configuration cycles across machines.
- Inconsistent language versions between student devices.
- Delays when labs are reset or locked down.
- Limited continuity of student workspace between sessions.

For beginner-focused tasks, the overhead of managing environment details can dominate actual programming time. That mismatch motivated the development of a browser-first workflow.

3.2 LIMITATIONS OF THE EXISTING SYSTEM

The limitations identified in baseline workflows are summarized below.

Limitations in Conventional Educational Coding Workflows

Limitation	Impact
Toolchain installation burden	New learners spend excessive time before first successful run.
Version inconsistency	Code that runs on one machine may fail on another.
Administrative restrictions	Lab systems may block installations and runtime updates.
Dependency drift	Environment state changes over time, causing unpredictable failures.
Remote execution dependency	Cloud systems require connectivity and backend reliability.

3.3 PROPOSED SYSTEM

NimbusCode proposes a local-first architecture where the browser itself is the runtime host. The core solution principles are:

- Keep execution client-side through WebAssembly runtimes.
- Persist workspace state in browser database (IndexedDB).
- Provide a familiar IDE-like interaction model.
- Support multiple languages through extension-based runtime mapping.
- Minimize deployment to static hosting requirements.

3.3.1 Functional Scope

The implemented scope includes:

- File and folder create/delete operations in an explorer tree.
- Tabbed editor workflow with Monaco integration.
- Runtime-aware execution from active file.
- Terminal output visualization and clear action.
- Settings tab for editor preferences (current and future extensibility).
- Local workspace persistence across browser reloads.

3.3.2 Non-Functional Scope

- Frontend-only architecture with no backend service dependency.
- Reasonable responsiveness on commodity hardware.
- Low initial onboarding complexity.
- Reproducible build and lint workflow.

3.4 ADVANTAGES OF THE PROPOSED SYSTEM

NimbusCode offers clear educational and engineering benefits:

- **Zero local compiler setup** for learners.
- **Consistent workflow** across operating systems with modern browsers.
- **Local privacy model** because source code need not leave the browser.
- **Low deployment complexity** through static hosting.
- **Extensible language architecture** through runtime mapping tables.
- **Immediate usability** for short labs and workshops.

3.4.1 Risk Register and Mitigation

Risk and Mitigation Summary

Risk	Potential Effect	Mitigation
Large bundle size	Higher initial load time	Apply code splitting and lazy runtime loading
Long-running code loops	Browser tab freeze	Add execution interrupt and timeout controls
Browser feature mismatch	Runtime not available on some devices	Validate capabilities and display fallback guidance
Single large component complexity	Maintenance cost increases	Modularize into feature components and hooks

4. SYSTEM DESIGN

Design Overview

The system is designed as a single-page application with explicit separation between presentation concerns, workspace persistence, and runtime orchestration.

NimbusCode High-Level Architectural Blocks

4.1 HIGH LEVEL DESIGN (ARCHITECTURAL)

Primary modules in the architecture:

- Explorer and editor interaction layer.
- Runtime selection and execution manager.
- File persistence adapter over IndexedDB.
- Output terminal bridge for run feedback.
- Settings and language metadata controls.

Data flow summary:

1. User selects file in explorer.
2. Active file opens in Monaco editor tab.
3. On run request, extension maps to runtime.
4. Runtime executes code and writes output to terminal.
5. File edits are debounced and persisted in IndexedDB.

4.1.1 Runtime Path for Interpreted Languages

For JavaScript, Python, PHP, Ruby, and SQLite, source text is sent to the relevant runtime through interactive execution API calls. Output streams are rendered in terminal UI. SQLite templates receive placeholder replacement before execution.

4.1.2 Runtime Path for Compiled Languages

For C and C++, NimbusCode runs an in-browser compile pipeline:

1. Build temporary in-memory filesystem with source file.
2. Execute clang in WebAssembly to produce object file.
3. Execute wasm-ld in WebAssembly to produce program.wasm.
4. Run resulting binary in WASI terminal context.

4.2 LOW LEVEL DESIGN

Low-level design details include path utilities, type models, workspace mutation logic, and UI handlers.

4.2.1 Type Models

```
export type WorkspaceFileEntry = {
  path: string
  kind: "file"
  content: string
  updatedAt: number
}

export type WorkspaceFolderEntry = {
  path: string
  kind: "folder"
  updatedAt: number
}

export type WorkspaceEntry = WorkspaceFileEntry | WorkspaceFolderEntry
```

4.2.2 Storage Access Layer

```
const DB_NAME = "nimbuscode-workspace"
const STORE_NAME = "entries"

const openWorkspaceDB = (): Promise<IDBDatabase> =>
  new Promise((resolve, reject) => {
    const request = indexedDB.open(DB_NAME, 1)
    request.onupgradeneeded = () => {
      const db = request.result
      if (!db.objectStoreNames.contains(STORE_NAME)) {
        db.createObjectStore(STORE_NAME, { keyPath: "path" })
      }
    }
    request.onsuccess = () => resolve(request.result)
    request.onerror = () => reject(request.error)
  })
```

4.2.3 Runtime Mapping Logic

```
const runtimeByExtension = {
  ".js": "quickjs",
  ".py": "python",
  ".c": "clang",
  ".cpp": "clangpp",
  ".php": "php-cgi",
  ".sql": "sqlite",
  ".rb": "ruby",
}
```

4.2.4 UI State Responsibilities

State variables coordinate selected file path, open tabs, expanded folders, run status, workspace readiness, error messages, and editor completion toggles. This central state approach improves feature velocity but increases component complexity, motivating future modularization.

5. System Architecture

5.1 Architectural Overview

NimbusCode follows a client-side-only architecture where all application components execute within the user's web browser. This design eliminates the need for backend infrastructure while providing a complete development environment. The architecture can be conceptualized in layers: the user interface layer, the application logic layer, the runtime abstraction layer, and the persistence layer.

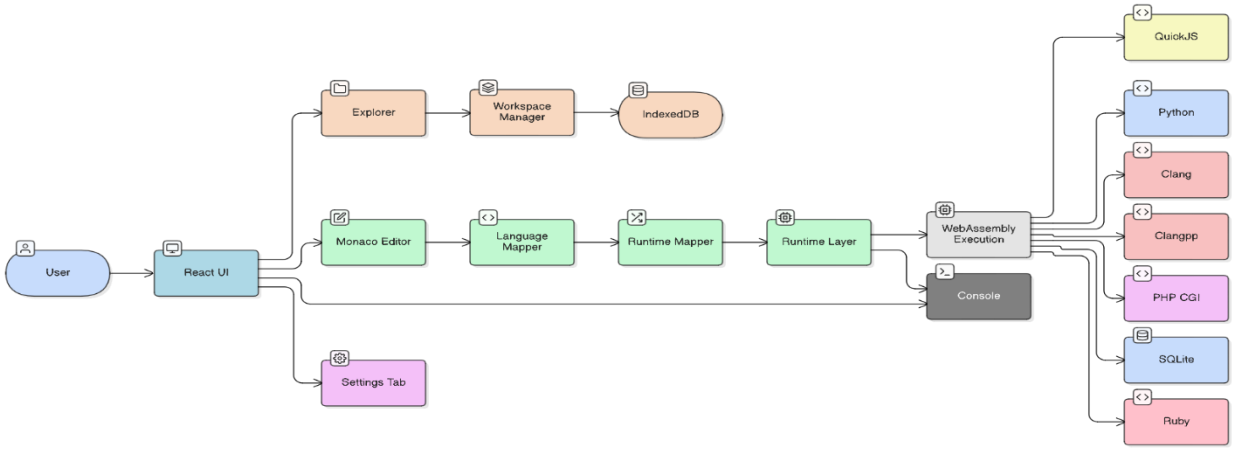
The user interface layer encompasses all visual components that users interact with directly. This includes the navbar, file explorer, editor tabs, code editor, and terminal output. These components are implemented as React components that render HTML elements styled with CSS.

The application logic layer contains the core functionality that coordinates between the UI and the underlying systems. This includes file and folder management, code execution orchestration, theme application, and completion provider registration. In NimbusCode, this layer is primarily contained within the App.tsx component.

The runtime abstraction layer provides a unified interface for executing code in different programming languages. This layer uses the @runno/runtime library to load appropriate WebAssembly binaries and manage execution contexts. For compiled languages like C and C++, this layer also coordinates the compilation process.

The persistence layer handles storing and retrieving workspace data using IndexedDB. This layer provides functions for listing workspace entries, saving new or modified files, and deleting files and folders. The persistence layer ensures that workspace data persists between browser sessions.

5.2 Component Interaction Flow



Understanding how data flows through the system helps clarify the architecture. When a user creates a new file, the following sequence occurs:

The user clicks the "+File" button in the explorer, which triggers the beginCreateEntry function with "file" as the argument. This function calculates the appropriate parent folder based on the currently selected item and sets the pending creation state, causing an inline input field to appear in the file tree.

When the user types a filename and presses Enter, the commitPendingCreation function is called. This function validates the input, creates a new WorkspaceFileEntry object with appropriate default content based on the file extension, and updates the entries state.

The new entry is immediately displayed in the file explorer and editor. In the background, the putWorkspaceEntries function is called to persist the new file to IndexedDB. The function opens the IndexedDB database, creates a transaction, and stores the new entry.

When the user edits code in the editor, the onEditorChange callback is triggered. This function updates the entry in the entries state and schedules a deferred save to IndexedDB using a debounce mechanism (250ms delay). This approach prevents excessive writes to IndexedDB during rapid editing.

When the user clicks the Run button, the runCode function initiates code execution. The function first determines the appropriate runtime based on the file extension, then either runs compiled code (for C/C++) or calls interactiveRunCode with the selected runtime and source code. The runtime loads any necessary WebAssembly binaries, executes the code, and streams output to the terminal component.

5.3 State Management Approach

NimbusCode uses React's built-in state management through the useState hook rather than external state management libraries. The main App component maintains the following state variables:

The entries state holds all workspace files and folders as an array of WorkspaceEntry objects. This state is the source of truth for the file system displayed in the explorer and the content shown in the editor.

The selectedPath state tracks which file or folder is currently selected in the explorer. This selection determines which item is highlighted and, for files, drives which content is loaded into the editor.

The activeFilePath state identifies which file is currently open in the editor. This may differ from selectedPath when the user has selected a folder but is editing a file in a different location.

The openTabs state maintains an array of file paths that are currently open in editor tabs. This enables users to switch between multiple open files without losing their place.

The expandedFolders state tracks which folders are expanded in the explorer tree view. This allows the explorer to show nested directory structures while allowing users to collapse folders they're not working with.

Additional state variables track runtime status (isRunning), user interface preferences (settingsCompletionsEnabled, settingsKeybinding), and various other aspects of the application state.

5.4 Module Organization

While the current implementation places most application logic in a single App.tsx file, the code demonstrates clear modular organization through the use of constants sections, helper functions, and type definitions. The file structure indicates the following logical divisions:

Constants section (lines 39-241): Contains all static configuration including runtime mappings, language definitions, file icons, code templates, and Monaco theme definitions. This section defines the static data that drives the application's behavior.

Completion providers (lines 243-433): Contains the code for registering language-specific autocomplete functionality with Monaco Editor. This includes both the completion item definitions and the functions that register providers with Monaco.

Compiled runtime support (lines 485-620): Contains the buildCompiledCommands function that constructs the compilation pipeline for C and C++ code, including the clang and wasm-ld command configurations.

Helper functions (lines 622-748): Contains pure utility functions for path manipulation, file system operations, and data transformation. These functions are used throughout the component to perform common operations.

Main App component (lines 751-1835): Contains the React functional component and all its hooks, callbacks, and rendering logic.

6. METHODOLOGY

6.1 REQUIREMENT ELICITATION AND ANALYSIS

Methodology began with practical pain-point analysis from educational coding contexts. Functional requirements were captured around four core needs: immediate usability, multi-language support, persistent workspace, and visual workflow familiarity.

Non-functional requirements focused on frontend-only architecture, reproducible builds, and predictable runtime behavior.

6.1.1 Requirement Matrix

Requirement Matrix

ID	Requirement	Implementation Evidence
FR-1	Create/delete files and folders	Explorer actions and IndexedDB operations
FR-2	Edit code with syntax support	Monaco editor configuration and language mapping
FR-3	Run active file by runtime mapping	Run button workflow and runtime tables
FR-4	Persist workspace between reloads	IndexedDB store with path-keyed entries
NFR-1	No backend for execution	Browser-only runtime usage
NFR-2	Static deployability	Vite output and static hosting compatibility

6.2 SYSTEM IMPLEMENTATION APPROACH

Implementation followed an incremental strategy:

1. Establish base project scaffold and lint/type safety.
2. Build explorer and tabbed editor interaction.
3. Integrate persistence APIs for workspace continuity.
4. Integrate runtime execution pipeline per language class.
5. Add UX refinements, settings tab, language metadata, and completion providers.

6.2.1 Iterative Deliverable Model

Each iteration targeted a testable output. This reduced integration risk and made debugging easier. By validating each layer independently (storage, editor, runtime), the project avoided high-cost late-stage rewrites.

6.3 VALIDATION AND VERIFICATION PROCESS

Verification combined static and dynamic checks:

- TypeScript strict-mode validation through build workflow.
- ESLint checks for code quality and hook correctness.
- Functional testing of file lifecycle and runtime execution.
- Manual regression across language mappings.
- UI interaction validation for tabs, explorer tree, and console output.

6.3.1 Build/Lint Verification Snapshot

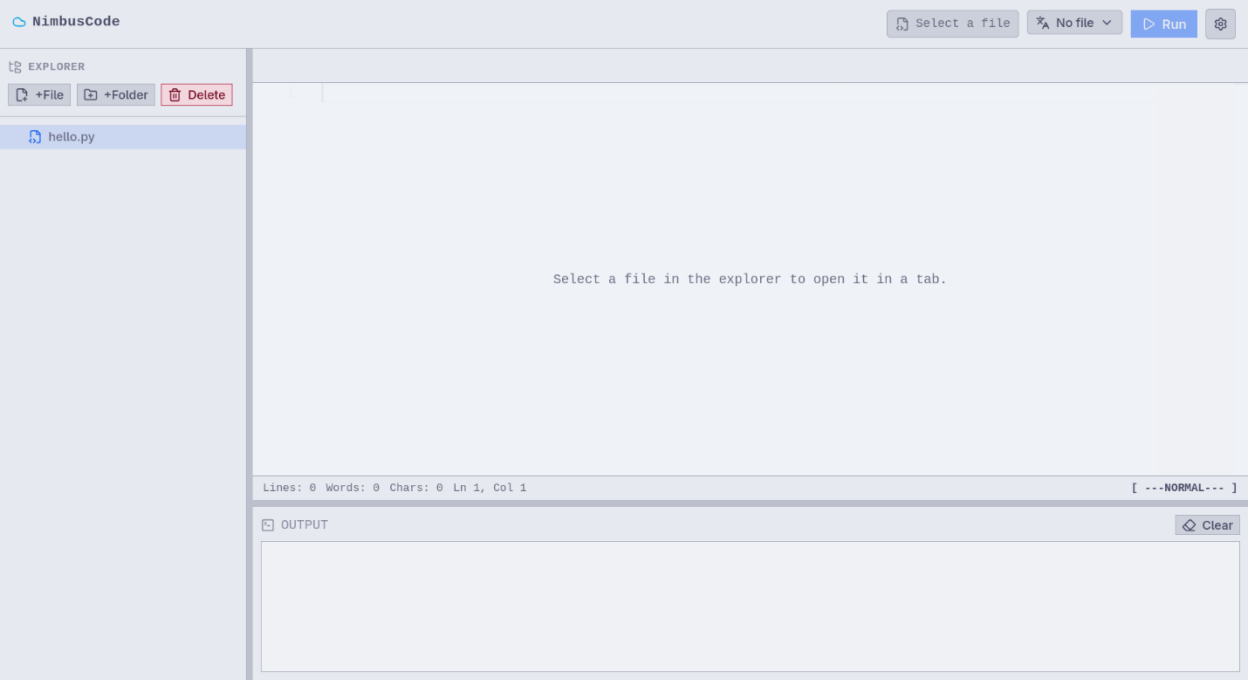
```
$ bun run lint
$ bun run build
vite v7.x building for production...
built in a few seconds
warning: chunk size exceeds recommended threshold
```

6.4 DOCUMENTATION AND REPORTING METHOD

Documentation was maintained in markdown and translated to structured report format. Technical notes were grouped into architecture, implementation, testing, and future scope sections. This report consolidates those materials into college-project documentation format.

6.5 RESULTS

6.5.1 The main user interface of NimbusCode



The interface is divided into four primary regions:

1. Top Navigation Bar
    - Displays the application title *NimbusCode* on the left.
    - Contains file selection dropdown, run button, and settings control on the right.
    - The "Run" button indicates runtime execution capability directly from the browser environment.
2. Explorer Panel (Left Sidebar)
    - Labeled as EXPLORER.
    - Provides buttons for:
        - +File (Create new file)
        - +Folder (Create new folder)

        - Delete (Remove selected entry)
    - This panel represents the virtual workspace tree stored in IndexedDB.
    - Currently empty, indicating no files have been created yet.
3. Editor Workspace (Center Panel)
    - Large central area intended for Monaco Editor integration.
    - Displays placeholder text: *"Select a file in the explorer to open it in a tab."*
    - This reflects the tab-based editor activation flow described in the System Design section.
    - No file is currently active, so the editor is in an idle state.
4. Output Terminal Panel (Bottom Section)
    - Labeled as OUTPUT.
    - Includes a "Clear" button for resetting terminal output.
    - This section displays runtime execution results (stdout/stderr).
    - Currently empty because no program has been executed.

6.5.2 Python Execution

When the Run button is clicked, NimbusCode maps the .py file to the Python WebAssembly runtime. The interpreter is initialized inside the browser and the script is executed line by line without compilation. When input() is encountered, execution pauses and waits for user input from the terminal. The entered value is passed back to the running Python process. Output generated by print() is streamed to the terminal panel. After the script finishes execution, the Python process exits and the runtime session ends.

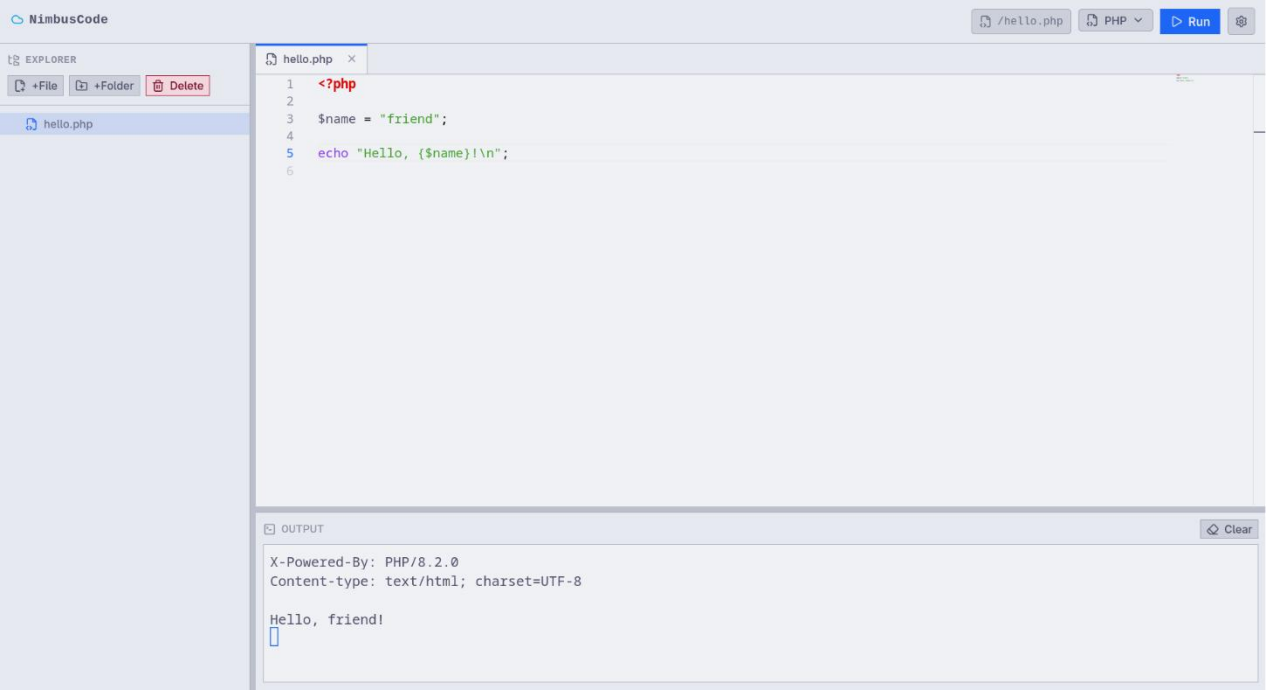
6.5.3 C++ Execution

When the Run button is clicked, NimbusCode maps the .cpp file to the C++ WebAssembly toolchain. The source code is first compiled using clang++ inside the browser to generate an object file. This object file is then linked using wasm-ld to produce a final WebAssembly binary (program.wasm). The generated binary is executed in a WASI sandbox environment. Standard input and output (std::cin and std::cout) are connected to the IDE's terminal panel. After execution completes and return 0; is reached, the program exits and the runtime session ends.

6.5.4 C Execution

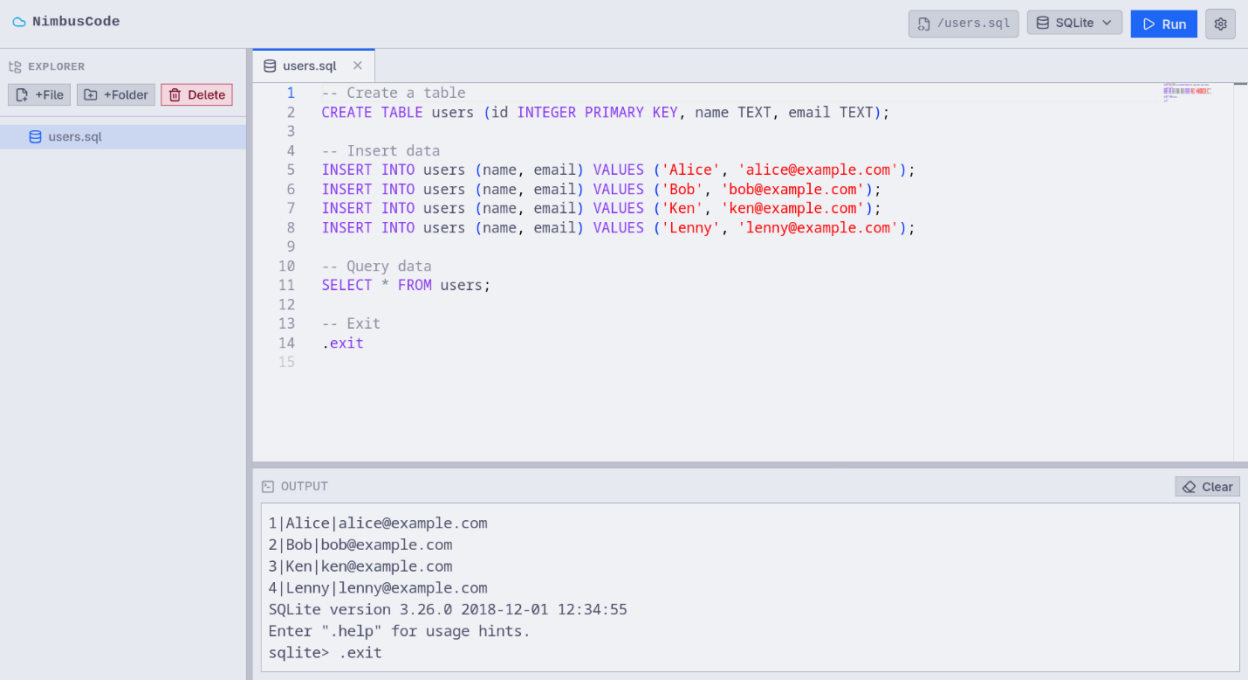
When the Run button is clicked, NimbusCode maps the .c file to the C WebAssembly toolchain. The source code is compiled using the in-browser clang compiler to generate an object file. This object file is then linked using wasm-ld to produce a WebAssembly executable (program.wasm). The generated binary runs inside a WASI sandbox environment. Standard output from printf is redirected to the IDE's terminal panel. After return 0; is executed, the program exits and the runtime session ends.

6.5.5 PHP Execution



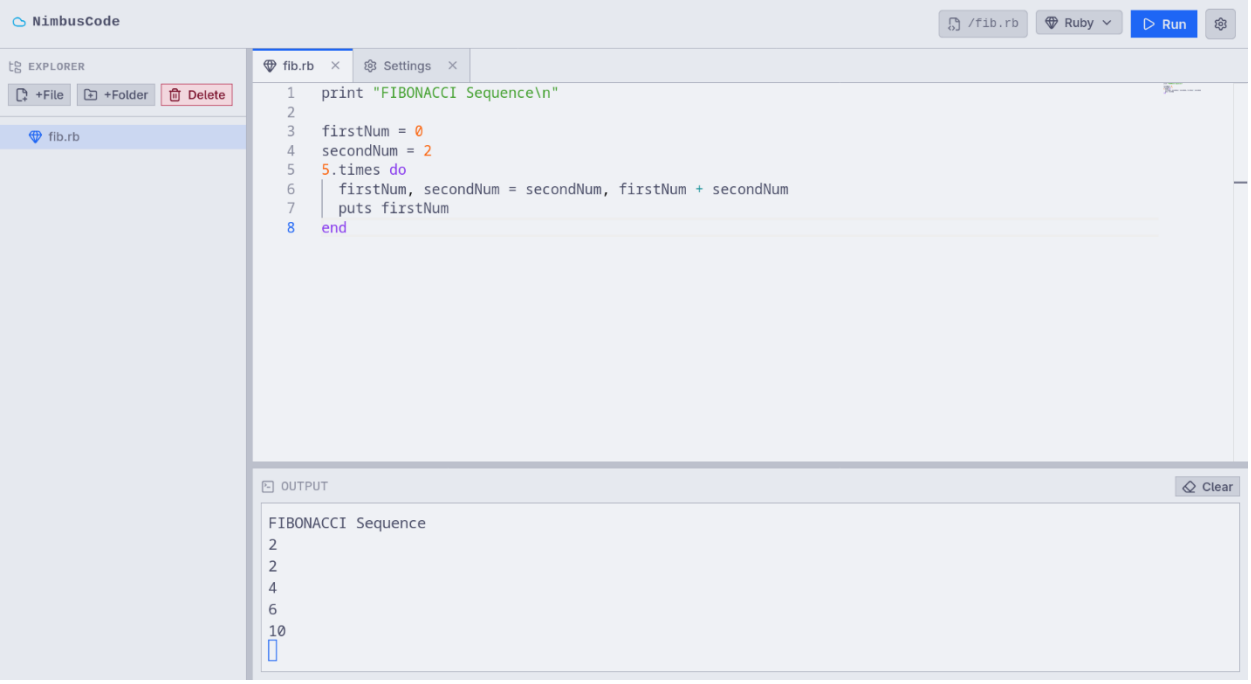
When the Run button is clicked, NimbusCode maps the .php file to the PHP WebAssembly runtime (php-cgi). The PHP interpreter is initialized inside the browser and the script is executed directly without compilation. The runtime processes the PHP file in CGI mode, generating HTTP-style headers along with the script output. Standard output from echo is streamed to the IDE's terminal panel. After the script finishes execution, the PHP process exits and the runtime session ends.

6.5.6 SQL Execution



When the Run button is clicked, NimbusCode maps the .sql file to the SQLite WebAssembly runtime. An in-browser SQLite engine is initialized and an in-memory database session is created. The SQL script is executed sequentially, processing table creation, insert statements, and the select query. Query results are formatted and streamed to the terminal output panel. SQLite CLI metadata (version and prompt) is displayed as part of the runtime session. When .exit is executed, the SQLite process terminates and the session ends.

6.5.7 Ruby Execution



When the Run button is clicked, NimbusCode maps the .rb file to the Ruby WebAssembly runtime. The Ruby interpreter is initialized inside the browser and the script is executed sequentially without compilation. The runtime evaluates the file directly in an isolated WASM sandbox environment. Standard output from print and puts is streamed to the IDE's terminal panel. After the script completes execution, the Ruby process exits and the runtime session ends.

6.6 Deployment

The NimbusCode frontend was deployed on Vercel and is publicly available at `https://nimbuscode.vercel.app`. The deployment process used Vercel's Git-based workflow: after connecting the repository, Vercel automatically installed dependencies, built the Vite app, and published the `dist` output as a static site. A key requirement for this project was enabling cross-origin isolation headers so browser runtimes using `SharedArrayBuffer` work correctly in production, which was handled through `vercel.json`.

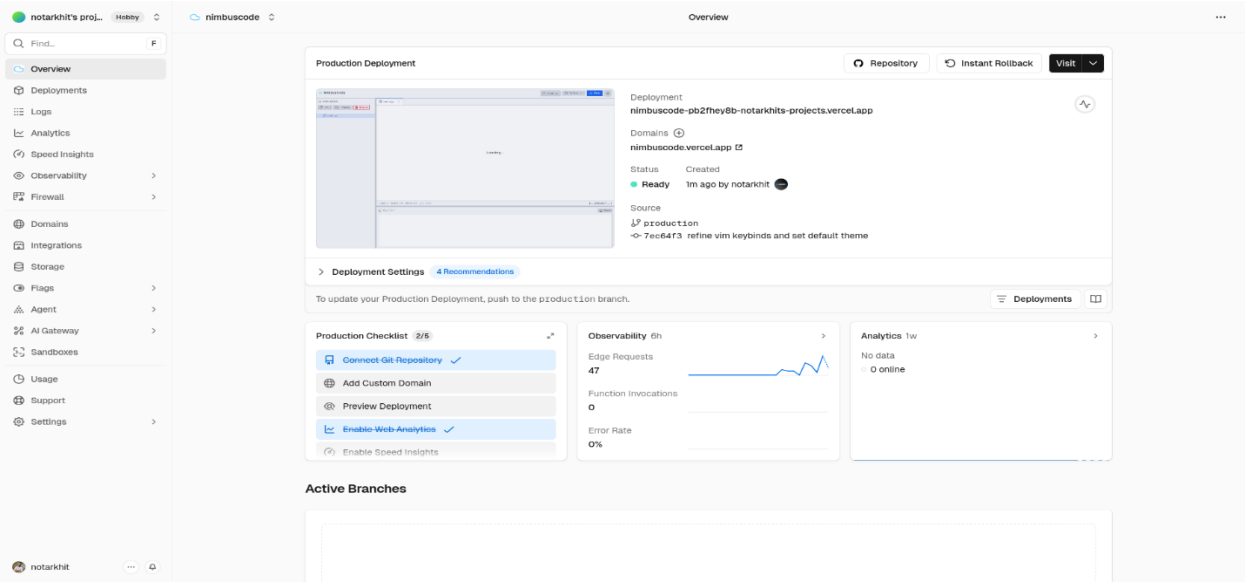
vercel.json

```
{
  "headers": [
    {
      "source": "/(.*)",
      "headers": [
        { "key": "Cross-Origin-Opener-Policy", "value": "same-origin" },
        { "key": "Cross-Origin-Embedder-Policy", "value": "require-corp" }
      ]
    }
  ]
}
```

build before deployment

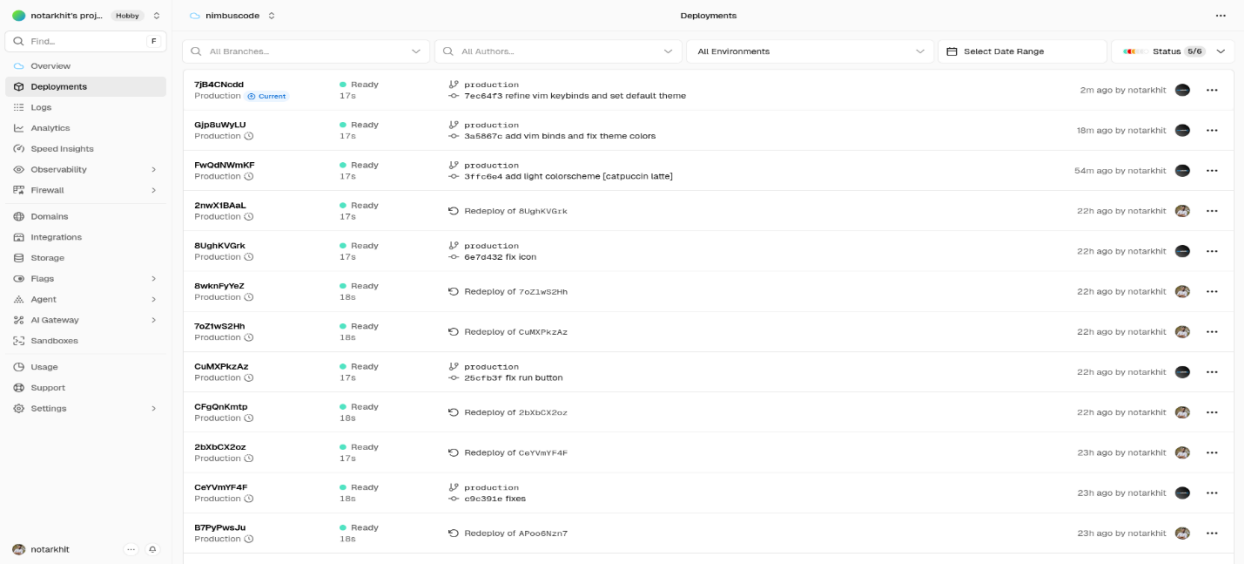
```
{
  "scripts": {
    "build": "tsc -b && vite build"
  }
}
```

6.6.1 Vercel-Overview



The image shows the Vercel dashboard overview page for the project named "nimbuscode," displaying a successful production deployment with the status marked as Ready and the live domain nimbuscode.vercel.app. The top section includes options such as Repository, Instant Rollback, and Visit, while the left sidebar contains navigation items like Deployments, Logs, Analytics, Domains, and Settings. A production checklist indicates some completed steps including enabling web analytics. The Observability panel shows edge request activity, and the Analytics section currently displays no data, indicating the project has been recently deployed.

6.6.2 Vercel-Deployments



The Deployments page of the Vercel dashboard for the "nimbuscode" project, listing multiple production deployments in a chronological format. Each deployment entry displays a unique build ID, environment labeled as Production, and a status indicator marked Ready with a green dot, confirming successful builds. The most recent deployment is tagged as Current and was triggered from the production branch with a commit message related to refining Vim keybinds and setting the default theme. Several entries indicate redeployments of previous builds, showing version iteration and testing cycles. Deployment durations are shown beside each entry, typically around 17–18 seconds, indicating fast build times. Author information and timestamps such as "2m ago" or "22h ago" appear on the right, providing traceability of changes. The left sidebar includes navigation options like Overview, Deployments, Logs, Analytics, and Domains, emphasizing project management and monitoring capabilities within Vercel.

7. TESTING

Testing focused on functional correctness, persistence behavior, runtime stability, and user-flow consistency.

7.1 Test Strategy

- **Unit-level reasoning tests:** Path normalization, runtime mapping, and state transitions.
- **Integration-level tests:** Explorer-to-editor-to-runner flow.
- **Persistence tests:** Browser reload and workspace restoration.
- **Runtime tests:** Language-specific code execution and output checks.

7.2 Representative Test Cases

Representative Functional Test Cases

TC	Scenario	Steps	Expected Result
TC	**Scenario**	**Steps**	**Expected Result**
TC1	Create Python file	Create file main.py, type code, open tab	File appears in explorer and editor tab
TC2	Run Python code	Click run on active .py file	Output appears in terminal
TC3	C compile pipeline	Create main.c, run code	Compile then execute within terminal
TC4	Delete folder recursively	Create nested folder and files, delete parent	Parent and child entries removed
TC5	Persist workspace	Reload browser tab	Files/folders restored from IndexedDB
TC6	Unsupported extension handling	Open unsupported file and run	User-visible runtime mapping error
TC7	Clear console action	Produce output, click clear	Terminal resets content panel
TC8	Completion toggle	Disable completions in settings	Suggestions/triggered completion disabled
TC9	Tab close behavior	Open multiple files and close active tab	Next fallback tab becomes active
TC10	SQL placeholder substitution	Run SQL with {{name}} token	Token replaced with safe literal and query runs

7.3 Testing Results Summary

Testing Outcome Summary

Area	Status	Notes
Workspace file operations	Passed	Create/edit/delete flows stable in manual tests
Persistence through reload	Passed	IndexedDB entries restored successfully
Runtime execution mapping	Passed	All mapped languages executed with expected routing
Compiled language pipeline	Passed	C/C++ compile and execute path completed
Build and lint checks	Passed	Strict TypeScript and ESLint checks succeeded
Performance optimization	Partial	Large bundle warning observed; needs code splitting

7.4 Identified Defects and Improvements

- Settings text currently says controls are placeholders while completion toggle affects behavior.
- Keybinding mode selector UI exists but has no active runtime behavior mapping.
- Initial production bundle size is larger than recommended threshold.
- No automated test suite integrated yet; validation is currently mostly manual.

7.5 Acceptance Criteria Review

The project meets core acceptance criteria for frontend-only operation, language execution support, persistence, and workflow usability. Additional criteria for advanced editor behavior and testing automation are captured as future improvements.

8. CONCLUSION

NimbusCode demonstrates that a practical educational coding environment can be delivered entirely as a frontend application. By combining a familiar IDE workflow with in-browser runtime execution, the project reduces setup friction and enables immediate learning-focused productivity.

The technical implementation validates multiple engineering goals:

- Multi-language execution without backend infrastructure.
- Local persistence through IndexedDB.
- Structured runtime flow for interpreted and compiled languages.
- Static deployment readiness with reproducible build pipeline.

From a pedagogy perspective, the system is suitable for introductory labs, workshops, and demonstrations where rapid onboarding is critical. From a software engineering perspective, the project can evolve into a larger platform through component modularization, automated tests, and advanced editor integrations.

Future Scope

- Implement actual Vim/Emacs keybinding integration.
- Add execution interruption control for long-running programs.
- Introduce lazy loading and manual chunk optimization.
- Add import/export workspace snapshots.
- Add optional sync mode while preserving local-first default.
- Introduce unit/integration test suites in CI.

9. BIBLIOGRAPHY

1. React Documentation. https://react.dev/
2. TypeScript Handbook. https://www.typescriptlang.org/docs/
3. Vite Documentation. https://vite.dev/guide/
4. Bun Documentation. https://bun.sh/docs
5. Monaco Editor. https://github.com/microsoft/monaco-editor
6. Runno Runtime. https://github.com/taybenlor/runno
7. WebAssembly Official Concepts. https://webassembly.org/docs/
8. MDN Web Docs: IndexedDB API. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
9. W3C Cross-Origin Isolation guidance and browser compatibility notes.
10. Course notes and institutional lab guidelines for web systems projects.

10. APPENDIX – Sample Source Code/Pseudo Code

Appendix A: Pseudo Code for Workspace Loading

```
function loadWorkspace():
   persisted = listWorkspaceEntries()
   if persisted is empty:
      source = initialWorkspace
      putWorkspaceEntries(source)
   else:
      source = persisted

   firstFile = first entry in source where kind == "file"
   setEntries(sort(source))
   setSelectedPath(firstFile.path)
   setActiveFilePath(firstFile.path)
   setOpenTabs([firstFile.path])
```

Appendix B: Pseudo Code for C/C++ Runtime Pipeline

```
function runCompiled(runtime, code):
   fs = { sourceFilePath(runtime): code }

   for step in prepareCommands(runtime):
      if step requires base filesystem:
         fs += fetchBaseFilesystem(step.baseFSURL)

      result = WASI.start(step.binary, args=step.args, fs=fs)
      if result.exitCode != 0:
         return failure("Compile step failed")
      fs = result.fs

   binary = readBinary(fs, "/program.wasm")
   if binary not found:
      return failure("No wasm produced")

   runResult = terminal.run(binary)
   return runResult
```

Appendix C: Core Storage Module Excerpt

```
export const putWorkspaceEntries = async (entries) => {
  if (entries.length === 0) return
  const db = await openWorkspaceDB()
  try {
    const transaction = db.transaction("entries", "readwrite")
    const store = transaction.objectStore("entries")
    for (const entry of entries) {
      store.put(entry)
    }
    await transactionDone(transaction)
  } finally {
    db.close()
  }
}

export const deleteWorkspacePaths = async (paths) => {
  if (paths.length === 0) return
  const db = await openWorkspaceDB()
  try {
    const transaction = db.transaction("entries", "readwrite")
    const store = transaction.objectStore("entries")
    for (const path of paths) {
      store.delete(path)
    }
    await transactionDone(transaction)
  } finally {
    db.close()
  }
}
```

Appendix D: Engineering Notes and Traceability

The following appendix pages document iteration-level progress logs, engineering observations, and deliverable checkpoints. These pages provide traceability for project execution across planning, implementation, validation, and documentation phases.

Appendix E.1: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 1 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 1. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 1 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.2: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 2 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 2. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 2 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.3: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 3 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 3. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 3 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.4: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 4 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 4. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 4 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.5: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 5 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 5. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 5 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.6: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 6 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 6. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 6 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.7: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 7 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 7. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 7 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.8: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 8 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 8. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 8 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.9: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 9 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 9. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 9 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.10: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 10 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 10. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 10 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.11: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 11 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 11. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 11 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.12: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 12 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 12. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 12 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.13: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 13 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 13. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 13 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.14: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 14 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 14. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 14 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.15: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 15 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 15. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 15 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.16: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 16 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 16. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 16 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.17: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 17 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 17. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 17 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.18: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 18 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 18. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 18 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.19: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 19 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 19. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 19 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.20: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 20 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 20. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 20 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.21: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 21 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 21. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 21 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.22: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 22 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 22. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 22 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.23: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 23 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 23. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 23 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.24: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 24 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 24. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 24 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.25: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 25 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 25. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 25 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.26: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 26 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 26. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 26 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.27: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 27 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 27. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 27 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.28: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 28 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 28. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 28 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.29: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 29 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 29. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 29 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.30: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 30 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 30. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 30 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.31: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 31 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 31. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 31 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix E.32: Iteration and Progress Log

Iteration Goal: Stabilize NimbusCode iteration 32 for academic reporting
Primary Tasks Completed: Reviewed explorer interactions, editor state behavior, runtime execution flow, and persistence checks for iteration 32. Updated implementation notes and synchronized source observations with the project report.
Observations and Outcomes: Iteration 32 improved documentation completeness, clarified known limitations, and preserved reproducible verification steps for build, lint, and runtime validation.

Deliverables recorded for this iteration:

- Updated source modules and refactored client-side workflow.
- Regression checks through manual run/lint/build verification.
- Documentation updates reflecting architectural and testing changes.
- Risk log updated with mitigation notes and pending improvements.

Status: Completed and documented for college project submission.

Appendix F: Glossary of Terms

Project Glossary

Term	Meaning in This Project
Term	**Meaning in This Project**
WebAssembly (Wasm)	Portable binary instruction format used for in-browser execution.
WASI	WebAssembly System Interface for sandboxed system-level interactions.
Runtime Mapping	Association of file extension with the execution runtime.
IndexedDB	Browser-native structured storage used for workspace persistence.
Workspace Entry	Data entity representing either a file or a folder path.
Explorer Tree	Left panel that displays folder and file hierarchy.
Terminal Panel	Output area where runtime stdout/stderr is displayed.
Completion Provider	Monaco integration that provides language suggestions/snippets.
Compile Pipeline	Multi-step process for C/C++ code to produce executable Wasm.
Cross-Origin Isolation	Header-based browser security mode enabling shared memory features.
COOP	Cross-Origin-Opener-Policy header used for isolation.
COEP	Cross-Origin-Embedder-Policy header used for isolation.
Static Hosting	Deployment model where built frontend assets are served without backend logic.
Debounced Save	Delayed persistence write to reduce frequent storage transactions.
Tab Activation Flow	Logic for selecting fallback tabs when active tab is closed.

Appendix G: Additional Test Artifact Matrix

Extended Manual Validation Matrix

ID	Validation Item	Result	Remarks
ID	**Validation Item**	**Result**	**Remarks**
M1	Open language dropdown from navbar	Pass	Supported list visible and dismisses on blur
M2	Select folder in explorer tree	Pass	Ancestor expansion state maintained
M3	Create nested folder path using inline input	Pass	Parent folders auto created where needed
M4	Create file from selected folder context	Pass	Path joined relative to selected directory
M5	Update file content and wait save debounce	Pass	Entry persisted after timeout
M6	Delete selected file with open tab	Pass	Tab removed and fallback activated
M7	Delete selected folder recursively	Pass	Child entries deleted from store
M8	Run JavaScript snippet	Pass	Output printed in terminal
M9	Run Python snippet	Pass	Prompt/input workflow available
M10	Run PHP program through php-cgi runtime	Pass	Program output displayed
M11	Run Ruby script	Pass	Runtime output displayed
M12	Run SQLite query with placeholder token	Pass	Token replacement path executed
M13	Run C code via compile/link pipeline	Pass	Binary generated and executed
M14	Run C++ code via compile/link pipeline	Pass	Binary generated and executed
M15	Trigger runtime mapping error on unknown extension	Pass	Error message shown in terminal
M16	Toggle completion switches in settings	Pass	Suggest settings updated in editor options
M17	Change keybinding select option	Pass	UI state changes (behavior mapping pending)
M18	Clear terminal output and rerun	Pass	Fresh output panel observed
M19	Reload page with existing workspace	Pass	Files restored from IndexedDB

ID	Validation Item	Result	Remarks
M20	Production build check	Pass	Build completes with large chunk warning

Appendix H: Extended Discussion on Maintainability

NimbusCode currently consolidates most UI and orchestration logic in a single large component. This helps rapid prototyping but increases cognitive load for future contributors. A maintainability-focused refactor should split responsibilities into dedicated modules such as ExplorerController, RuntimeRunner, EditorTabs, WorkspaceService, and SettingsPanel.

A modular architecture would improve testability and reduce accidental coupling between view-state transitions and runtime events. Hooks can encapsulate behaviors like tab lifecycle, persistence debounce, and completion provider registration. This report recommends refactoring in phases to avoid behavior regressions.

Appendix I: Deployment Checklist

6. Install project dependencies with bun install.
7. Verify lint and type checks using bun run lint and bun run build.
8. Ensure static host serves COOP/COEP headers where required.
9. Deploy generated dist/ files to static hosting platform.
10. Validate runtime behavior in target browser set.
11. Confirm IndexedDB persistence and permission behavior.
12. Capture post-deployment smoke test evidence.

Appendix J: Sample Command Log

```
$ bun install
$ bun run dev
$ bun run lint
$ bun run build
$ bun run preview
```

Appendix K: Final Remarks for Submission

This report is prepared in the requested structure and style for college project submission. It includes architectural reasoning, implementation evidence, testing artifacts, bibliography, and appendices that support both technical review and academic evaluation.