ChatGPT

RedByte OS & Logic Playground – Product and Systems Specification

Abstract

RedByte OS is a browser-based **desktop environment and simulation platform** built to teach digital logic by letting users literally construct a computer from scratch. Its flagship application, **Logic Playground**, provides an interactive circuit sandbox where learners progress from wiring a lamp to designing a simple CPU. The system emphasizes immediate visual feedback, local-first operation (no cloud or login required), and a seamless blend of open-ended exploration with structured educational guidance. We present the product vision, core design principles, architecture, user experience philosophy, and technical implementation plan for RedByte OS and Logic Playground, culminating in a phased roadmap and an implementation directive. This document serves as the authoritative specification for engineering, ensuring the final build meets all defined goals and quality bars.

Introduction

Modern logic simulators and educational tools often force a trade-off between **playful exploration** and **structured learning**. RedByte OS aims to eliminate this divide by providing an integrated "logic computing playground" where a 6th-grade beginner can flick on a virtual lamp in seconds, and an advanced student can gradually ascend to building a working CPU – all in one continuous environment. The guiding philosophy is that if a user can build a computer from first principles, they will truly understand it [1]. To achieve this, RedByte OS adopts a full OS-style interface (complete with boot sequence, desktop, and multi-window apps) to host Logic Playground and other tools. This approach brings a sense of realism and context to the learning experience, making the platform feel like a mini operating system dedicated to logic design.

In the following sections, we detail the **problem space** that RedByte addresses, the **vision** for the product's impact, and the concrete **design and system architecture** decisions that underpin its implementation. We outline how the user experience is crafted to cater both to freeform tinkerers and to learners following a curriculum. We then delve into the **technical architecture** – a TypeScript monorepo with modular packages – that makes this possible, including a deterministic simulation engine and an offline-first state model. Key user workflows (the "North Star" user loop) and interaction models are described, along with performance considerations to ensure a lively, 60 FPS experience. A comprehensive testing strategy ensures the system remains robust and regressions are caught early. Finally, we present a phased **roadmap** from the current foundation to future enhancements, and a one-page **Implementation Directive** that translates this specification into actionable engineering tasks.

Problem Statement

Bridging the Learning Gap: There is a significant gap between playing with basic logic gates and understanding how real computers work. Traditional curricula jump from simple circuits to abstract CPU concepts without an intuitive bridge. Learners often struggle to connect the **1-bit world** of AND/OR gates to the **32-bit world** of processors. Existing tools are fragmented: one might use a breadboard simulator for gates, a separate IDE for assembly, and textbook diagrams for CPUs. This fragmentation makes it hard for learners to see the continuity. **RedByte's goal** is to provide a single continuous learning environment that scales with the user's understanding [1] – from a single wire lighting a lamp to a complete (simulated) computer – thus demonstrating that "computers are understandable" and built from the same logic pieces they started with.

Lack of Interactive Feedback: Many learning tools are either static (diagrams in a book) or slow feedback (writing HDL code and running it). Beginners benefit from **immediate visual feedback** – seeing a lamp light up or a gate output change *as soon as* they flip a switch. Without this, the connection between cause and effect in logic is weakened. RedByte addresses this with a live simulation that updates in real-time as the user manipulates the circuit, reinforcing understanding through instant feedback [2].

Exploration vs. Instruction Dilemma: Pure sandbox tools (e.g. open-ended circuit simulators) are great for exploration but offer little guidance, while guided tutorials can be restrictive and stifle curiosity. The problem is how to allow open exploration without letting novices get lost, and how to provide guidance without annoying experienced users. RedByte OS solves this by splitting concerns: Logic Playground is always available as a **sandbox** with no forced tutorial prompts, and a separate **Help App** provides structured lessons only when invoked. There are no modal tutorial overlays in the Playground itself (a deliberate non-goal) – users are never forced down a path when they just want to tinker. Conversely, those who seek a curriculum can opt into guided tracks in the Help App, ensuring both personas (the explorer and the student) are satisfied.

Inconsistent and Non-deterministic Simulations: Some circuit tools do not guarantee determinism – e.g. event-driven simulations where race conditions might cause different outcomes, or physics-based simulators that require real-time clock speed. For an educational tool, **deterministic behavior** is essential: given the same circuit and inputs, the results should always be the same [3]. This predictability is crucial for learning (and testing). RedByte's logic engine is designed as a discrete tick-based simulator to ensure deterministic updates and reproducible outcomes for every circuit tick [4] [3]. This also simplifies stepping through circuit behavior step-by-step, another requirement for teaching complex sequential logic.

Offline Access and Data Ownership: Many modern tools require cloud accounts or internet connectivity, which can be barriers in classroom settings or for younger students. RedByte OS is **local-first**: it requires no login or connection. All features (saving circuits, running simulations, etc.) work offline and data is stored locally (in the browser's storage or as files) so that schools and individuals retain control. This addresses privacy concerns and ensures no critical learning moment is interrupted by connectivity issues. A side benefit is zero installation friction – users can just open the web application (or a desktop-packaged version) and start using it without installs, with the system booting to a usable state in under 3 seconds [5].

User Interface Fragmentation: In many DIY or open-source simulators, the UI/UX can be inconsistent or clunky (e.g. different dialogs with different styles, or some actions only accessible via mouse). This can confuse learners. RedByte aims for a polished, coherent UI grammar: consistent design language (colors,

typography, component styles) across the OS and apps, and a **keyboard-centric** interaction model so power-users can perform any action without leaving the keyboard. Every feature – from opening files to wiring components – has a keyboard shortcut or accessible UI path, avoiding "dead ends" that require a mouse (an explicit global contract of the design). This not only improves accessibility but also builds good habits (similar to how an IDE encourages learning shortcuts).

In summary, the problem RedByte OS tackles is how to create a **unified, engaging, and rigorous** environment for learning logic and computer organization, one that scales with the user's skill, provides instant feedback, and respects the user's autonomy (offline, no lock-in). The following sections describe how our product vision and system design directly address these challenges.

Product Vision

Computers, demystified. The vision of RedByte OS is to become the **canonical platform for learning and experimenting with digital logic**, where any user – be it a middle-schooler or an engineering student – can start from the very basics and, through a series of intuitive steps, end up constructing a working computer system. By hosting this experience in an OS-like environment, we make the journey feel comprehensive and realistic: the user isn't just coding or wiring in a vacuum, they are "booting" a system, launching apps, and ultimately creating a computer within a computer. This meta-computing notion (a computer building another computer) is the endgame of the vision, reinforcing the message that all complex computers are built from simple principles.

Our North Star is the idea that *any motivated person can understand how computers work, given the right tools*. RedByte OS + Logic Playground is that toolset. It's akin to a flight simulator for computer architecture: providing a safe, controlled environment to perform feats that would be impossible or impractical in real hardware, and to learn by doing.

Concretely, the product vision manifests as: - **A seamless learning curve:** Users progress through "layers" of complexity (we define a 9-layer hierarchy from wires up to meta-computation [6]) without abrupt jumps. Each layer's concepts build directly on the previous, and the environment evolves with the user. For example, a user might start by playing with a virtual battery and lamp (Layer 0), then combine gates to form an ALU (Layer 2), then add flip-flops to create stateful circuits (Layer 3), and so on, up to designing a simple CPU (Layer 6) that can actually run inside the simulator. Throughout, the *same* interface and tools support them – they don't have to switch platforms when they outgrow basics. - **Empowering creativity and insight:** By providing both an exploratory sandbox and structured guidance, the platform encourages users to try their own ideas (What happens if I connect these gates in a weird way?) while also ensuring they grasp key concepts (Why does that SR latch behave that way?). The environment celebrates "aha!" moments – for instance, when a user accidentally creates a known circuit, the system recognizes it (e.g., "You just built a Half Adder!") and offers to name or save it [7]. This reinforces learning and gives positive feedback, turning passive recognition into active engagement. - **Broad appeal and longevity:** The product is envisioned not just as a one-time tutorial but as an ongoing playground and reference. A 6th grader might use it to learn basics for a science fair, a hobbyist might use it to prototype a logic idea, a college student might use it to do homework on CPU design, and a professor might use it live in lectures to demonstrate concepts. By combining an OS shell (which can host multiple apps) with the core Playground, RedByte can extend to related domains (e.g. a "Logic Help" app with lessons, a "Files" app for managing circuit files, even potential future apps like a "Logic Quiz" or a community circuit browser). The vision is a platform that can

grow and adapt, potentially even allowing community contributions (e.g. shared circuit libraries or custom components, via optional future cloud features).

Finally, at its most ambitious, RedByte OS could support "computer inception": a scenario where the user-built CPU in Logic Playground is capable of running a program that in turn designs new circuits (layer 8 of our roadmap). Achieving that will validate the entire progression and offer a profound narrative moment to the user: a computer they built is now building a computer *inside* of itself [8]. This is the ultimate realization of our vision and a testament to the platform's power – showing that with the right approach, the once-daunting world of computer architecture becomes not just understandable, but exciting and creative.

Design Principles

Our design principles blend educational pedagogy with practical UX and engineering rules. They serve as inviolable guidelines throughout the project:

- **Immediate Visual Feedback:** The system should respond to user actions in real time. Whenever a user flips a switch, connects a wire, or changes a signal, they see the effect *instantly* – lamps glow, gates output change, counters increment, etc. This tight feedback loop keeps users engaged and reinforces cause-and-effect understanding [2]. For example, as soon as a wire connecting power to a lamp is placed, the lamp icon lights up (no need to "run" the circuit in a separate step). This principle also guided our simulation design (continuous tick engine) and is reflected in features like live-updating outputs and an always-on clock signal animation.

- **Progressive Disclosure of Complexity:** The UI and available features should start simple and gradually expand as the user is ready for more. Early on, a new user might see only basic components (wires, switches, lamps, a few gates) and an example or two. As they complete certain milestones or choose to explore further, more components and options become available (e.g. once they use a clock, timing controls appear; once they create a subcircuit, the hierarchy tools appear). This prevents overwhelming novices with too many options at once, while never artificially limiting advanced users. The Help App curriculum is structured around this principle, introducing concepts layer by layer (from simple gates to complex chips) [2]. In the UI, progressive disclosure is seen in things like layered example menus (organized by difficulty layer) and optional advanced panels (e.g. an **Oscilloscope** view or **Trace** viewer for timing analysis can stay hidden until needed).

- **Shareable by Default:** Anything a user builds can be easily shared with others **without special setup**. We implement one-click sharing that encodes the entire circuit into a short URL string [5] [9]. The user can press a Share button (or `Ctrl+Shift+C`) to copy a URL, which others can paste to load the identical circuit. This lowers the barrier for classroom collaboration and community sharing – no accounts or server needed, just a URL. Internally, this is achieved by serializing the circuit JSON and base64-encoding it into the URL hash or query param [10]. By making sharing so frictionless, we encourage users to show off their creations and teachers to distribute examples or assignments as links. It aligns with our educational mission: knowledge (and circuits) should flow freely.

- **Zero-Friction Access & Local-First:** RedByte OS is entirely **web-based and offline-capable**. The design target is that a new user can go from hitting the website (or launching the app) to interacting

with a circuit in under 3 seconds [5]. This means optimized loading (we modularize code and avoid heavy assets), no mandatory sign-ups or tutorials to click through, and an initial state that immediately shows something interesting (e.g. a boot animation that leads right to an open Playground window or a welcome demo circuit). All functionalities (saving, loading, using the Help App) work offline using the browser's local storage or file system – reinforcing the local-first principle. The user never *needs* to make an account or be online to use full features. This principle stems from recognizing that many educational contexts have limited internet or restrictive policies, and that users should own their data. The application will *never* wall off features behind cloud accounts (any online features we add in the future, like publishing, will be optional enhancements – see Non-Goals).

- **Keyboard-First, No Dead-Ends:** We design the UI such that any action that can be done with a mouse can also be done with keyboard (often faster). A rich set of keyboard shortcuts is available globally and within apps [11] [12]. For instance, a user can hit `Ctrl+O` to open the circuit file dialog, `Ctrl+S` to save, arrow keys to move focus between interface elements, and even connect wires via keyboard (by selecting a component's port and using arrow or enter keys to choose a target, or other accessible navigation – this is being considered). This principle is evident in the provided cheat-sheet (e.g. Undo/Redo with `Ctrl+Z/Y`, switching windows with `Alt+Tab`, launching apps or search with global shortcuts, etc.) [13] [12]. The rationale is twofold: (1) Efficiency for advanced users (reducing reliance on slow pointing device actions), and (2) Accessibility – users with motor challenges or who simply prefer keyboard control can fully participate. We ensure there are no "mouse-only" operations that block such users (for example, even drawing a wire can be canceled with `Esc` or done via keyboard selection as noted). A visible outcome of this principle is the **Command Palette** (`Ctrl+Shift+P`) which allows many actions to be executed by typing their name, and the comprehensive **Keyboard Shortcuts** help dialog available via `?` key.

- **Consistency & UI Grammar:** All interface elements across the OS and apps follow a consistent design language – a **UI grammar** – so that once a user learns one part, the rest feels familiar. This includes consistent color themes (governed by a token system in `rb-tokens`), typography, spacing, and control styles provided by a common library (`rb-primitives` for base components, and `rb-theme` for theming) [14] [15]. Modal dialogs look and behave the same whether they're a "Save As" in Playground or a "Settings" dialog in the OS shell. Icons are uniform and come from the same `rb-icons` set to avoid mismatched art styles [16]. We also standardize interactions: e.g., pressing Escape should consistently close the current modal or unfocus the current window (this is a defined OS-level behavior in the shell) [17]. Another example: all apps use the same window frame with minimize/close buttons and can be moved or resized similarly – the windowing system provides this uniform chrome. Error messages and toasts are phrased in a consistent, friendly tone. By having a coherent UI grammar, we reduce cognitive load; the user learns the "language" of the interface once and can apply it everywhere, focusing mental energy on the logic concepts instead of on how to use the software.

- **Deterministic, Reversible Operations:** Every action in the system should be predictable and (when applicable) undoable. The **determinism** aspect applies especially to the logic simulation – given a circuit and sequence of inputs over time, the simulation results are fully determined (no randomness or timing ambiguity) [3]. This also extends to state changes: for example, saving a circuit or creating a chip yields the same result every time, and no hidden state should cause one run to differ from another. The **undo/redo** aspect is crucial for a creative tool – users will experiment freely only if they

trust they can easily revert mistakes. Logic Playground maintains a history of edits (with a depth of ~50) and allows multi-step undo/redo via keyboard or menu [18]. Implementing this required designing all state mutations through central stores so that a history entry can capture a snapshot before each change. Deep cloning of circuit state is used to ensure that undo truly reverts to a prior state without lingering side-effects [19]. This principle of reversible operations not only aids UX but also testing – it implies a purity to our state transformations that makes automated testing easier (no irreversible operations means we can cycle through states in tests and end up consistent).

- **Educational Scalability:** As an educational product, RedByte OS is designed to scale not in terms of users or data, but in terms of concepts and user knowledge. The system's design always considers *the next step* for the learner. For instance, the file format for circuits is versioned and backwards-compatible so that as we introduce new component types or features, old saved circuits (from earlier lessons) will still load and function [20]. This avoids frustrating situations where a learner's earlier work (or a teacher's prepared example) breaks as the software evolves. Similarly, the architecture is modular so that we can introduce new capabilities (e.g. a 3D visualization module, or a new type of analysis tool) without reworking the whole system – thus the platform can grow with the curriculum. This principle is a bit meta, but it influences decisions like having placeholders for future packages (`rb-logic-3d`, `rb-logic-adapter`, etc.) [21], and designing the APIs in `rb-logic-core` to be extensible (e.g. a plug-in registry for new logic node types). The ultimate idea is that the platform will remain useful and up-to-date as pedagogy advances or as we tackle more ambitious teaching goals (like that meta-circuit generator in Layer 8).

These principles guided every aspect of the design and will continue to serve as touchstones during implementation and future enhancements. Any proposed change or new feature is evaluated against this list – if it violates any principle (without a very strong justification), it's reconsidered or redesigned.

UX Philosophy (Exploration vs. Education)

RedByte OS embraces a dual-faceted UX philosophy: it should be a **playground for exploration** and simultaneously a **guided tour for education**. The key to reconciling these is separation of contexts and user control over when to switch modes.

Exploration Mode – Logic Playground: When users are in the Playground app by itself, the experience is open-ended and non-prescriptive. The interface in this context emphasizes discovery and creativity. The user is free to pull out components from the palette, wire them up however they like, and press play. There are **no intrusive pop-ups or forced tutorials** interrupting them – this is an explicit design choice (no clippy, no "hint" arrows unless explicitly enabled). If a user just wants to experiment with crazy circuit ideas at 2am, Logic Playground will not nag them to follow a lesson. It's a sandbox in the truest sense.

To still facilitate learning during exploration, we provide **lightweight cues and celebrations** that don't feel like hand-holding. One such feature is the **pattern recognition system**: as the user plays around, if they happen to build a known useful circuit (say they arrange gates into an XOR pattern), the system will unobtrusively toast a message: *"You just built an XOR! Adds two bits with no carry."* [7]. This gives the user positive feedback and perhaps entices them to learn more about what they just stumbled upon. A "Save as Chip" button may also appear, letting them encapsulate that circuit for reuse (more on that in a moment). These toasts and prompts are **triggered by the user's actions** (and only when a significant pattern is

detected), so they feel like achievements rather than instructions. They never block the user from doing something else – they appear and fade after a few seconds, consistent with the exploratory ethos.

The UI also supports exploration through features like **examples and templates** that the user can choose to load. The Examples menu is organized by complexity layer (with descriptions like "Layer 2 – Arithmetic" to hint at progression) [22]. But it's the user's choice to load an example; the app won't load one for them without asking. When an example is loaded, it's presented as just another circuit the user can poke at and modify, not a read-only demo – exploration is always encouraged.

Furthermore, **multi-modal views** in Playground encourage exploring circuits from different angles. A user in exploration mode might open the Oscilloscope panel to see voltage traces, or switch to a schematic logic diagram view to see a more abstract representation of their circuit. These tools are available for them to self-serve deeper insight as curiosity leads. The environment is rich enough to support tangents: if a user wonders "what if I turn this sub-circuit into a reusable chip?", they can do so and then perhaps open multiple instances of it to see how modular design works – all without leaving exploration mode or needing permission.

Education Mode – Help App (Guided Learning): When the user opts into the Help application, they essentially step into a guided learning experience that runs in parallel to the Playground. The Help App provides structured content (lessons, challenges, context) under the banner of Tracks A, B, C (detailed in a later section). The UX challenge is to integrate this guidance without compromising the freedom of the Playground. We achieve this by treating the Help App as just another window in the OS. It can sit side by side with Playground: e.g., the user has Playground open on one side and a Help App lesson on the other, each as separate windows. The Help App might say: "Try placing an AND gate and a OR gate and observe the difference in outputs," at which point the user does so in Playground. The key is that *the user* initiates each step – the help content suggests and the user executes. There is no automatic script or modal wizard in Playground performing actions for them.

The Help App and Playground communicate in minimal, intentional ways. For example, if a lesson refers to a specific example circuit, clicking a link in the Help App can send an **intent** to Playground to load that example (the underlying system uses the OS intent mechanism – a controlled way to have apps interact [23] [24]). This respects app isolation and still gives a smooth experience: the user doesn't have to manually find the example; it appears in their Playground window when they request it. But even this is user-initiated. If the user wants to ignore the guide and do something else, they can – the Help App won't yank control away.

The **split between exploration and education is also reflected in UI tone and elements**. In Playground (exploration), UI elements are oriented towards creation (toolbox, wiring, simulation controls). In Help App (education), the UI is text and multimedia – possibly with an occasional quiz or prompt. We consciously avoid overlaying instructional arrows or highlights onto the Playground canvas itself; instead, the Help App might say "Notice how the lamp turned on – if not, check your wire connection," rather than the Playground popping up a "connect this here" arrow. This keeps the contexts psychologically separate: one window is "my creation space," the other is "my guidebook." It's akin to having a lab instructions handout next to your breadboard in a real electronics lab.

Encouraging the Switch: The UX does provide gentle nudges to engage with the educational content for users who might benefit. For instance, on first launch, after the boot-up, the desktop might show a **"Logic Help" icon pulsating** subtly or a **Welcome overlay** that says "New to logic? Open the Help app for guided

tutorials" (with an easy keyboard shortcut shown). This appears exactly once or only when no user data is present, so as not to annoy returning users. The idea is to let novices know the resource is there. Once dismissed, the Playground is entirely theirs until they choose otherwise.

Spatial and Mental Model: The OS metaphor helps maintain a clear mental model: Playground and Help are two separate apps. This means a user can literally minimize/close the Help window if they want to "focus" on just exploring, and later reopen it from the launcher if they want to resume a track. This is much less frustrating than a one-size-fits-all UI that tries to integrate tutorials into the main canvas (which often leads to either intrusive pop-ups or so much subtlety that users miss the guidance). Instead, we leverage multi-window UX to cleanly separate modes.

Summary: The UX philosophy can be summarized as *"Guide when asked, otherwise get out of the way."* Exploration is always possible, encouraged by subtle feedback systems (like pattern recognition to name discoveries and undo to enable wild experimentation without fear). Education is available at any time, deeply thought out as a parallel experience that complements the exploration. By keeping the two modes separate-but-cooperative, RedByte ensures that a user can fluidly move between **play** and **learn** as they see fit. This dual approach maximizes both engagement and educational efficacy, catering to self-driven discovery and structured learning in one unified platform.

System Architecture (Monorepo Structure & OS–App Boundary)

RedByte OS Genesis is implemented as a **TypeScript monorepo** organized into discrete packages, each responsible for a specific facet of the system. This modular architecture provides clear boundaries between the OS (shell) functionality and the application-specific logic, while allowing all parts to live in one codebase for easy integration and consistency. Below is a high-level overview of the monorepo structure and how responsibilities are divided:

```
/packages
  @redbyte/rb-shell       — The OS shell (desktop environment, boot sequence) [25]
  @redbyte/rb-windowing   — Window management system (multi-window state, z-index, focus) [26]
  @redbyte/rb-apps        — Application registry and launcher (defines how apps are registered and opened) [27]
  @redbyte/rb-logic-core  — Core logic simulation engine (circuit model, tick simulation) [28]
  @redbyte/rb-logic-view  — 2D circuit rendering components (canvas drawing of gates/wires) [29]
  @redbyte/rb-logic-3d    — (Planned) 3D visualization module (for future enhancements like 3D chips) [21]
  @redbyte/rb-logic-adapter — Adapter between core and view (utilities for syncing engine state to UI)
  @redbyte/rb-icons       — Icon library (SVG react components for UI icons: gates, window buttons, etc.) [16]
  @redbyte/rb-theme       — Theming system (light/dark themes, CSS variables via React context) [15]
```

```
   @redbyte/rb-tokens       – Design tokens (consistent colors, spacing,
typography scales) [30]
   @redbyte/rb-primitives   – Basic reusable UI components (buttons, modals,
etc., ensuring consistent UI grammar)
   @redbyte/rb-utils        – Utility library (shared helpers, e.g. storage
handlers, math, etc.)
/apps
   playground/              – The Logic Playground application (React-based UI,
bundled via Vite) [31]
   (future apps like Help, Files, etc., would also live under /apps)
```

(Note: some packages like rb-logic-3d, rb-primitives are placeholders set up in the monorepo for future development, ensuring the project scaffold accounts for them [21].)

OS vs App Boundary: In the above structure, the **OS** can be thought of as the combination of `rb-shell`, `rb-windowing`, `rb-apps`, and some shared subsystems (theme, tokens, icons). The **flagship application** (Logic Playground) primarily lives in `apps/playground` and leverages logic-specific packages (`rb-logic-core`, `rb-logic-view`) as well as OS services.

- **RedByte OS (Shell & Windowing):** The shell (`rb-shell`) is responsible for the overall container application – it boots up with an animated sequence, presents the desktop UI (maybe a background, a dock or start menu with apps), and manages the lifecycle of windows. It provides global features like the **BootScreen** (with the progress bar and orb animation on startup) [32], global keyboard shortcuts (e.g. open launcher, switch windows, etc.) [11], and cross-app services like a centralized **Toast** system for notifications [33]. The windowing package (`rb-windowing`) handles things like creating new windows, tracking their positions, focus, minimize/maximize state, etc. Essentially, the OS layer abstracts a mini operating system: one can launch multiple apps, move windows around, and use OS-level commands. It also likely includes a **File system abstraction** (`useFileSystemStore`) which acts as a local filesystem for the Files app and for apps to load/save files (e.g., Playground uses it to save `.rblogic` circuit files in a "Documents" folder) [34] [35]. The OS ensures that multi-window and multi-app interactions follow certain contracts (like singleton apps vs multi-instance apps, focus behaviors) as defined in design (for example, the "Launcher" app may be a singleton that opens with Ctrl+K, and Files app can have multiple windows) [36] [37]. The OS is also where global state like user settings persist (theme choice, simulation speed default, etc., often via `rb-utils` and localStorage) [38].

- **Logic Playground App:** The Playground is implemented as a React app in the `apps/playground` directory, but it's not a standalone website – it's packaged to be launched within the RedByte OS shell. The `rb-apps` registry provides metadata for the Playground (name, icon, maybe a component to render) and the shell knows how to launch it in a window [27]. Internally, Playground uses the logic engine (`LogicEngine` class from rb-logic-core) for simulation and the logic view components for rendering the circuit canvas. The Playground's code is structured into UI components (palettes, canvas, inspectors), state stores (for history, for chips, for tutorials, etc.), and integration with OS services like file I/O and toasts [39] [35]. The **boundary** between OS and Playground is clean: the OS doesn't need to know about circuit internals, it just hosts the window. Communication happens via explicit channels (for example, when the user hits "Save" in Playground,

Playground calls `useFileSystemStore` from `rb-apps` to save data – which under the hood might use OS-level localStorage API, but through a controlled interface). Conversely, if the OS wants Playground to do something (say open a specific file because the user double-clicked a `.rblogic` file in the Files app), it will invoke Playground's app through an intent, providing the file ID, and the Playground app code handles loading that file into its state [40] [35]. This decoupling ensures that, for instance, we could upgrade the Playground app or swap it out, and as long as it adheres to the same app API (open, close, handle intents), the OS layer remains unaffected.

- **Dependency Flow:** By design, **applications depend on the OS and on domain-specific libraries, but not vice versa**. The package dependency graph reflects this: `rb-apps` (where app definitions and logic live) depends on `rb-shell` and `rb-windowing` (because apps might call shell APIs to open windows, etc.) and also on `rb-logic-core` / `view` (for Playground's domain) [41]. However, `rb-shell` does not depend on `rb-logic-*` – it is generic. And `rb-logic-core` is completely standalone (it can run without React or any browser UI, making it testable in isolation) [42] [43]. This separation improves maintainability: the OS can evolve its features (like new window management tricks or a new app like Settings) without touching Playground internals, and the logic engine can be optimized or replaced without the OS caring (as long as Playground updates to use it).

- **State Management:** Each package and app has its own state management appropriate to its scope, but at a high level we use a combination of React hooks (for local component state), Zustand-like stores or context for cross-component state (for things like circuit data, history, or OS-level window list), and browser persistence for long-term state (localStorage for user settings, etc.). For example, Playground uses a `historyStore` to track undo history in an immutable way [18], a `chipStore` to manage saved chips (persisted to localStorage under a namespace like `rb:chips`) [44] [45], and a `hierarchyStore` to manage the stack of circuits for drill-down (not persisted, just in-memory while the app runs) [46]. The OS has a `windowStore` (likely inside rb-windowing) that tracks open windows globally [47], and a `settingsStore` for things like the global tick rate setting [47] [48]. Communication between these states is done in controlled ways: e.g., Playground might call `setWindowTitle` via the windowStore to change the OS window's title when a circuit is loaded [49], or the OS shell might listen to changes in windowStore to render the taskbar/dock. Because of the monorepo, these shared stores can be imported across package boundaries as allowed by the dependency rules (the tsconfig and package JSON enforce that apps can depend on core packages but not the other way around) [43].

- **File Format & Compatibility:** As part of the architecture, we defined a **circuit file schema** (with versioning) in `rb-logic-core`. A circuit (saved as `.rblogic` JSON) contains a list of nodes and connections, plus a version tag [20]. The core provides serialization/deserialization functions [50]. This means both Playground and any other app (say a future circuit analyzer) can use the same core library to read/write circuit files, ensuring consistency. The OS (via the Files app) treats these like documents. By keeping the schema versioned (currently version "1"), we have an architectural guarantee of backward compatibility: future changes must either extend the schema in a compatible way or bump the version and include a converter for old files. This was an explicit design decision to support the educational use-case of longevity – e.g., a teacher's circuit file from last year should open in the latest version of Playground without issues, even if internally we added new node types (they would just be ignored or stubbed if not present in the old file, or auto-upgraded if possible).

In summary, the system architecture separates **concerns** cleanly: the OS shell deals with user interface paradigms (windows, launching apps, global shortcuts, persistence, theming), while the Logic Playground app deals with domain specifics (circuit editing, simulation). They communicate through well-defined interfaces (app registry, intents, and shared stores where appropriate). The monorepo structure (with ~11 core packages and a few apps) reflects this modular design, enabling parallel development (e.g., working on the simulation engine while someone else refines the window manager) and ensuring that each part can be tested in isolation. This architecture sets a strong foundation for adding features – for example, adding a new app (like the Help App or a Settings App) doesn't require entangling with the logic engine, it just plugs into the shell and possibly reuses some primitives (like modals or icons). Conversely, enhancing the logic engine (say adding a new gate type or performance improvement) is done in `rb-logic-core` and does not affect OS-level code except that Playground will benefit from it. The clear OS/App boundary also makes it feasible to consider deploying Playground in other contexts (for instance, a standalone mode inside an iframe or an Electron wrapper) by providing a minimal shim of the shell – but within RedByte OS, we leverage the full OS paradigm for a richer experience.

Data & State Model (Local-First, Deterministic, Undo-Safe)

The data model of Logic Playground is built around a **local-first, persistent state** approach with a strong emphasis on determinism and full undo/redo capabilities. We describe the key aspects: how circuit data is represented, how it's stored and manipulated, and how state changes are managed safely.

Circuit Data Representation: At the heart is the `Circuit` data structure defined in `rb-logic-core`. A Circuit consists of a collection of **nodes** (logic components like gates, switches, etc.) and **connections** between node pins [51] [52]. Each `Node` has a unique ID, a type (which determines its behavior, e.g. "AND" or "FlipFlop"), a position (for UI rendering), and optionally some internal state (for stateful nodes like flip-flops which might store their last output) [51]. A `Connection` is a simple pair of endpoints (`from` node's output pin to `to` node's input pin) [52]. This data structure is purposefully minimal and serializable – it captures the essential topology of the circuit. When a circuit is saved or shared, this is exactly what gets persisted (with a version tag as discussed) [20]. The decision to keep nodes and connections as separate lists (rather than, say, embedding connections in nodes) ensures we can easily add/remove connections and treat the circuit as a graph in the simulation engine.

Deterministic Simulation State: The simulation engine (`CircuitEngine`) uses the Circuit definition to perform logic evaluation. Importantly, the engine does not add extra randomness or hidden state; it processes all nodes in a defined order (such as topologically or by a stable ID ordering) each tick, producing the same outputs given the same inputs every time [53] [4]. Any stateful behavior (like a clock oscillating or a latch remembering a value) is handled via explicit node types (e.g., a `Clock` node toggles its output every tick according to a known pattern, and a `Delay` node introduces a one-tick delay to break combinatorial loops) [54] [55]. This means that the *only* source of nondeterminism would be user input (e.g., when the user toggles a switch at an arbitrary time) – and even that is made deterministic in playback by quantizing to simulation ticks. In other words, if a user toggles a switch, the effect is applied on the next simulation tick boundary, ensuring we can replay it in tests or in conceptual time without ambiguity.

Local-First Persistence: All data – circuits, chips, settings, etc. – lives on the user's machine. For persistence, we have a couple of mechanisms: - **LocalStorage** is used for small pieces of data that should persist automatically. For example, the theme preference is stored so the OS boots with the last chosen

theme, and the chip library is stored so that user-created chips persist across sessions [44]. The chip system in fact uses a key like `rb:chips` in localStorage to save the array of custom chip definitions [56]. Each chip definition (name, subcircuit data, etc.) can also be exported/imported as JSON for backup or sharing [57] [58], but the primary store is the browser. - **IndexedDB or File System API:** For larger data like circuits, RedByte OS includes a pseudo-file-system accessible via a Files app. When the user saves a circuit in Playground (`Ctrl+S`), it goes through the FileSystemStore which likely uses IndexedDB under the hood to store the file content by an ID, and maintains a directory structure (we simulate a basic filesystem). The user perceives this as saving a `.rblogic` file in, say, a "Documents" folder. This design means the circuit files are available offline and are not tied to any cloud service. (Under the hood, using IndexedDB or localStorage for file bytes is an implementation detail; the key is that it's within the user's browser profile or device storage). - **In-Memory State and Autosave:** While editing, the current circuit lives in React state (or a store) within Playground. We maintain a flag `isDirty` to indicate unsaved changes [59]. An autosave mechanism periodically (debounced) writes the circuit to a backup (localStorage or similar) so that if the app or page crashes, the user doesn't lose work – on next load, Playground checks for a backup and offers recovery [60]. This was implemented to improve resilience.

Because everything is local, there's zero reliance on network latency or server uptime for the core experience. This aligns with our offline requirement and also means state reads/writes are extremely fast (no HTTP overhead).

Undo/Redo and Immutability: The Playground editor's state model treats the circuit as an immutable object for the purposes of history. Each time the user makes an edit (places a gate, draws a wire, deletes something, etc.), a new Circuit object is produced and pushed onto a history stack [18]. The `historyStore` holds up to 50 such states and provides `undo()` and `redo()` functions that simply move a pointer backward or forward and set the current circuit state accordingly [18]. To avoid accidental mutation (which would break our undo), we perform *deep cloning* or pure function updates. For example, adding a gate is implemented as `currentCircuit = produce(currentCircuit, draft => { draft.nodes.push(newNode) })` – using something like Immer or manual cloning – so that the previous state remains untouched [19]. This guarantees that when we revert to a prior state, it's exactly as it was.

The undo/redo system is also tied into the UI: Undo and Redo menu items are enabled/disabled based on `canUndo` / `canRedo` observables from the store [35], and performing an undo triggers a UI refresh (since the circuit object the canvas is rendering has changed). We also decided that certain actions clear the history for sanity – e.g., loading a new file will reset the undo stack (because undoing into a previous file's state could be confusing if the user conceptually treats files as separate workspaces) [61]. And conversely, actions like "New Circuit" essentially just create an empty circuit state and clear history.

Deterministic Tick and Simulation State: The simulation has its own state, primarily the current output values of each node (and internal states for sequential nodes). However, we do not persist or long-retain simulation state separate from the circuit structure. If you close a circuit and reopen it, all flip-flops start cleared, all clocks start from 0, etc. This is a conscious choice to keep things simple and deterministic (the user can always set up initial conditions via input switches or known states). Within a session, the simulation state is encapsulated in the `CircuitEngine` instance. Each tick deterministically updates that state. If needed for debugging, we could snapshot this as well (e.g., to implement a "traveling wave" timeline or a deterministic replay of signal changes, we could record input changes with tick indices – something possibly for a future trace feature).

Global State and Isolation: At the OS level, certain global states exist (window positions, app focus, etc.), but these are isolated from app logic. Each app maintains its own state for content. This prevents accidental bleed-over (for example, an undo in Playground won't undo something in the Help app, because they have separate history contexts; and the OS doesn't even know what an "undo" means in Playground aside from routing the Ctrl+Z keystroke to it when that window is focused). The OS does keep a global list of open windows and their last-known state (for session restore – e.g., if you refresh the page, it could restore the windows that were open and maybe even which file each Playground was showing) [62]. This is handled by serializing the window state to localStorage on every significant change (window opened/closed/moved) [63]. The OS's global state (like settings or window list) is also simple JSON data that's persisted, and at boot the shell will rehydrate from it.

No Duplicate Sources of Truth: We follow an architectural invariant that each piece of data is owned by a single module/store to avoid split-brain scenarios. For example, the current circuit in Playground is held in one place (likely the hierarchy store when not drilling down, or a top-level React state) [64], and everything references that. We do not keep multiple copies of the circuit data in different components; components subscribe to the single source. Similarly, the list of user-defined chips is solely in `chipStore` and any UI that needs it (like the chip palette dropdown) pulls from there, rather than each maintaining its own list. This invariant is to ensure consistency and is part of our "no state duplication" rule for architecture clarity (see Definition of Done). It also makes undo trivial, as we just swap out the single source with a past version.

Local Schema Versioning: Because our data (circuit files, etc.) might live for a long time on users' machines, we planned for version migration. If in v2 we introduce, say, a new property on nodes or a new type of connection, we will bump the file version to "2" and provide a `deserialize` that can read v1 and upgrade it (or at least handle missing fields gracefully) [20]. The system will likely always be backward compatible at least one version or more, or we'll supply a migration tool. This way, a curriculum written using v1 won't break when v2 of the software is deployed – critical for trust in educational deployments.

In short, the data and state model in RedByte is engineered to be **predictable, robust, and user-friendly**. Everything lives on the client side under the user's control, changes are explicit and reversible, and the simulation behaves like a pure function of the current state (plus tick count) – ideal for both teaching and testing. This deterministic local-first approach not only boosts performance (no latency to servers) but aligns perfectly with our principle of enabling fearless exploration: the user can try anything, save at will, undo if needed, and know that the circuit will behave the same tomorrow as it did today.

Core User Loop: *Build → Observe → Chip → Dive → Reflect*

At the core of the user experience is a cyclic workflow we consider our **North Star** for engagement and learning. We want users continually moving through this cycle, each time at a slightly higher level of understanding or complexity. The stages of the loop are: **Build, Observe, Chip, Dive, Reflect.**

- **Build:** The user actively constructs something – e.g., places gates and wires to form a circuit that achieves a goal (even if the goal is just "see what happens"). In this stage, the user is in creation mode: dragging components from the palette, wiring outputs to inputs, labeling things, perhaps resizing the canvas for convenience. The system supports the build stage with a smooth UI (drag and drop, click to connect, intelligent wiring assists) and real-time validation (if there's something

obviously wrong like a short circuit or floating input, we might subtly highlight it or log a warning, but not stop the user). Building is where the user's hypothesis or idea takes shape tangibly.

- **Observe:** Once a circuit (or a portion of it) is built, the user needs to test and observe its behavior. This is the experimental stage. The user might turn on the simulation (if it's paused) or step through clock cycles. They toggle input switches, change constant values, or even use interactive input components (like a "button" node that they can press). They watch what happens: do the lamps light up? Does the output equal the expected truth table? Maybe they open the Oscilloscope view to watch a timing diagram of signals over several ticks. The system facilitates observation by providing multiple ways to visualize state: LEDs on the schematic change color, output digits on a 7-seg display node update, the trace viewer might list signal changes, etc. Importantly, observation is often *concurrent* with building – in Logic Playground, you typically don't need to enter a separate run mode; the circuit can be "live" as you build (with a running clock, etc.). This immediate feedback during observation ties closely to our principle of immediate feedback.

- **Chip (Abstract):** This stage is about **encapsulation and re-use**. After building something that works (say a full adder circuit from gates), the user is encouraged to turn that sub-circuit into a reusable component – essentially, to create a new "chip" or block that they can treat as a black box. This is facilitated by the *Save as Chip* feature: when the system recognizes a known pattern or whenever the user chooses, they can select a group of components and save them as a custom chip [65]. They give it a name, define its input/output ports (the UI auto-suggests likely ports by analyzing which wires enter/leave the selection) [66] [67], and categorize it under a layer (e.g. "Layer 2" if it's an adder) [67]. Once saved, this chip appears in the component library (with a distinct icon/color) and can be dropped into other circuits [68]. The **Chip stage** of the loop represents the user taking ownership of their new understanding: they built something concrete, and by encapsulating it, they acknowledge "this is a building block I know how to make and I might use it to build bigger things." Technically, what happens is the circuit definition is stored and a new node type is registered on the fly in the LogicEngine representing that sub-circuit [69]. This abstracts away the details inside it during use, which reduces visual clutter and cognitive load in subsequent builds.

- **Dive:** Now that the user has a new "chip" or at least a completed circuit, the next step is to dive deeper or higher: *deeper* into complexity or *higher* in abstraction. This can take two forms:

- **Drill-Down (Understanding Existing Complexity):** If the user (or someone else, via an example or a provided library) has a complex chip, "diving" means they open it up to see how it works. RedByte OS supports a hierarchical view – the user can double-click a chip to **enter** its schematic, seeing the internal circuit that defines it, with a breadcrumb or navigation UI to go back up [70] [71]. This is literal diving: going down into the hierarchy. It's crucial for learning that any abstraction can be peeled open. For instance, a student who has been handed a "FullAdder" chip can dive in to see the two half-adders and OR gate that make it up, then return to the top-level view.
- **Next-Level Build (Applying Complexity to Build More):** Alternatively, "Dive" can mean using the newly acquired building block to construct something even more advanced – i.e., diving into the next challenge. For example, after making and chipping a full adder, the user might dive into building a 4-bit ripple-carry adder using multiple instances of their new chip. In effect, this is starting the loop over at a higher level: build (with chips now as primitives), observe its behavior (does the 4-bit adder produce the right sums?), maybe chip that as a "4-bit Adder", and so on. Each cycle through the loop moves the user one level up in the abstraction hierarchy.

The system supports the Dive stage by maintaining the hierarchical stack (so the user can always go back up to where they started) [70] and by providing UI cues (like breadcrumbs "Top Level > FullAdder > XORGate" indicating the path inside nested chips). Dive is also where the **Help App** or narrative can intervene with deeper insight: e.g., after the user successfully builds and uses a chip, the narrative Marcus might chime in with "Now you've treated your circuit as a single unit – that's exactly how engineers manage complexity!" reinforcing the concept of abstraction.

- **Reflect:** This is a quieter but vital stage where the user consolidates what they've done and learned. After building and experimenting, perhaps creating a new chip or reaching a milestone, the system encourages reflection. This could be through:
- **Narrative commentary:** The optional Marcus narrative might provide a short explanation or a thought-provoking note when the user achieves a key milestone. For example, upon first creating a reusable chip, Marcus might comment on the significance of abstraction. Or after building a working CPU, he might note: "Think about it – you've constructed a machine that can compute. All those little pieces came together to do something big."
- **Help App prompts:** The Help App might explicitly include reflection sections in its lessons, asking the user questions like "Why did we need a flip-flop instead of a latch here?" or "What do you observe about the pattern of outputs, and what does that imply?" The user can write down their thoughts or just internally answer them.
- **User's own actions:** The platform provides tools for reflection such as the ability to **save** the circuit and perhaps annotate it. A user might take a snapshot of the output (we might allow exporting a truth table or waveform) to analyze. The act of naming a chip itself is a reflective exercise (e.g., "What does this circuit essentially do? Let's name it appropriately.").

Reflection is often done outside the immediate flow of the app – e.g., the user might simply lean back and think, or discuss with a peer or teacher. Our role is to prompt and make it easy (e.g., by providing the narrative or by not rushing them to the next thing automatically).

After reflection, the idea is that the user either has a new goal or a new question, which naturally leads back to **Build** with a fresh perspective. For instance, having reflected on how an SR latch works and what its limitations are, the user might decide "I want to build a D latch next" – and they dive back into building that, starting the loop anew.

An example of one loop iteration: Imagine a user is learning about binary addition: - In **Build**, they connect two 1-bit adders (half-adders) together with an OR gate for the carry – essentially creating a full-adder circuit from scratch. - In **Observe**, they toggle the two input bits through all combinations (00, 01, 10, 11) and watch the sum and carry outputs on lamps. They see it matches binary addition truth tables. Perhaps they open an inspector that shows a truth table view updating live. - They proceed to **Chip** this circuit, naming it "FullAdder", with two inputs (A, B, Cin) and outputs (Sum, Cout). The system acknowledges the save with a toast and the new chip icon appears in their component library (maybe under a category "Custom" or "Layer 2"). - Now they **Dive** upward: they create a new blank circuit and in the Build stage of the next loop, they place four "FullAdder" chips in a row to make a 4-bit adder. They wire the carry-out of each to the carry-in of the next, etc. They test this with some inputs, observe it adds 4-bit numbers correctly. This might have been guided by the curriculum or their own curiosity. - They take a moment to **Reflect**: the user realizes that by using their component four times, they've scaled the adder to bigger numbers easily. Marcus might comment: "Notice how once you had a FullAdder, making a 4-bit adder was just repetition. That's the power of abstraction – you didn't deal with all the little gates again." The user internalizes this concept, feeling accomplished.

This loop keeps the user engaged and learning: each cycle adds to their toolkit (both mental and the literal component library via chips) and prepares them for the next challenge. The UI is designed to facilitate transitioning smoothly between these stages (e.g., toggling simulation on/off for observe, one-click to save chip, double-click to drill in, etc., so the user isn't stuck at any stage).

Continuous Loop with Curriculum: The Help App is aligned with this loop as well – often a lesson will guide the user through one or two iterations of the loop. For example, a lesson might explicitly say: *Build* this circuit, *Observe* what happens with these inputs, now *Reflect* on that outcome (maybe answer why it failed under a certain input), now *Build* the improved version, etc. By reinforcing this methodology, we're teaching the scientific method of circuit design within the tool.

In conclusion, the **Build→Observe→Chip→Dive→Reflect** loop is the engine of user progress in RedByte. Our design ensures that at any given moment, the user knows what stage they are in (or can naturally flow to the next), and the system provides the necessary features to support that stage. This loop cultivates a self-sustaining learning process: users create, witness results, abstract their knowledge, apply it to bigger problems, and think about what they learned, which then inspires further creation.

(We note that not every scenario will involve a formal "Chip" step – e.g., if a user is just exploring a single-layer concept, they might loop Build→Observe→Reflect→Build… until they have something worth encapsulating – but the presence of the chip mechanism means whenever they reach a meaningful milestone, they have that option to lock it in and ascend the abstraction ladder.)

Interaction Model (Keyboard-First, Smooth Wiring, Live Toggling)

The interaction model of RedByte OS and Logic Playground is crafted to feel **fluid, immediate, and powerful**. Users should feel like they are directly manipulating a live circuit with minimal friction. Key aspects include heavy use of keyboard shortcuts, intuitive wiring mechanics, and a live-edit simulation where changes take effect instantly.

Keyboard-First Interactions: As mentioned, practically every action has a keyboard shortcut. This goes beyond just the typical copy/paste or undo. For example: - The **Global OS shortcuts** allow quick navigation and system control: `Ctrl+Space` opens a Spotlight-like system search (to launch apps or find files) anywhere [11], `Ctrl+Shift+P` opens the command palette for advanced commands, `Alt+Tab` cycles through open windows (just like a desktop OS) [13], and `Ctrl+W` closes the focused window [72]. These create a desktop-like feel where a power user can manage multiple apps without touching the mouse. - **Playground App shortcuts:** Within Logic Playground, as soon as that window is focused, the keystrokes map to editor functions. For instance, `Delete` deletes the selected component(s), `Ctrl+Z/Y` undo/redo circuit edits [12], `Ctrl+O` opens the circuit file open dialog [73], `Ctrl+S` saves the current circuit [74], `Ctrl+Shift+C` exports a shareable link [75]. There are also likely shortcuts for sim control: e.g., pressing the spacebar toggles Run/Pause simulation (this is common in simulators and indeed our help overlay shows "Space – Run/Pause simulation" [76]). - **In-canvas hotkeys:** We plan and implement interactions where the keyboard aids circuit building. For example, multi-select is possible with Shift+Click (mouse+keyboard) or by holding Shift and using arrow keys to expand selection if we implement that [77]. Once multiple items are selected, one could use arrow keys (with maybe Alt or something) to nudge them, or press a key to rotate components if rotation were a feature (currently logic gates probably have fixed orientation, so not applicable yet). We might map number keys to place specific components (like 1 = AND

gate, 2 = OR gate, etc., as a quick palette, though this might conflict with typing into something else – needs careful design). - **Command Palette & Search:** The Command Palette (Ctrl+Shift+P) likely provides a quick way to execute any command via text, beneficial for advanced interactions (like “Toggle grid” or “Snap to grid on/off”, or “Focus next input field”). The global search (Ctrl+Space) can also be an interaction method; for instance, a user could type an expression like “AND gate” and maybe drag it from search results into the circuit (this is hypothetical but could be a future integration).

Ensuring no dead-ends, even tasks like wiring can partly use keyboard: e.g., after selecting a component, pressing Tab might cycle focus through its pins, and Enter might “pick up a wire” from a pin, then using arrow keys or search to select a target pin, and Enter to connect. This is an advanced keyboard feature; if not fully implemented, at minimum keyboard can select and delete wires by focusing them (ensuring accessibility).

Smooth Wiring: Wiring components together is one of the most frequent actions, and we focus on making it feel as “natural” as possible in a digital sense. When the user wants to connect components: - They can either **click and drag** from a pin: The moment they click on an output pin, the UI starts drawing a wire following the cursor. As they drag, the wire is rendered as a smooth curve or polyline (likely orthogonal segments on a grid). We ensure the wire rendering is at 60 FPS to follow the cursor without lag, and perhaps use a slight magnetic grid so that wires tend to run horizontally/vertically (for neatness) unless the user forces a diagonal. - Alternatively, the user can **click-release-click** (point-to-point): Click on a source pin, then the system highlights potential targets or enters a mode where the next click on a compatible target pin will complete the connection [77] . This is useful for users who prefer not to drag. The UI gives immediate feedback: when a pin is clicked, it might highlight that pin and all eligible target pins (maybe with a subtle glow) to indicate “click any of these to connect.” If the user changes mind, pressing Escape cancels the wire (as indicated in our help: Esc clears selection or cancels a wire in progress [78] [79]). - If a user attempts to draw a wire in an invalid way (like from an input pin to another input), the system will not complete that connection and may give a gentle feedback (like a shake or a red flash on the invalid target) but *will not crash or scold*. The smoothness here implies forgiving UI: you can start a wire and hit the wrong thing, just cancel or reattach it elsewhere easily.

We also consider **auto-routing** for wires to avoid tedious manual drawing: likely the engine auto-routes wires with default path (Manhattan style) between the two pins once connected. The user can then drag the wire or its midpoint if they want to reshape it (if advanced editing is allowed). Smooth wiring also extends to creating buses: possibly dragging a wire from a bus node attaches multiple parallel wires, though initial version may treat each bit individually.

Additionally, selecting wires should be easy (click on the wire path highlights it). We ensure that wires are selectable and deletable just like components, and that multi-select works (so you can lasso a bunch of gates and wires to move them together). All these contribute to a “smooth” editing experience where the user isn’t frustrated by fiddly interactions.

Live Toggling & Real-Time Interaction: The Logic Playground simulation is often running live as the user builds. Input nodes like switches or buttons can be toggled with a simple click (or key). For example, clicking on a switch graphic flips its state (0 to 1 or vice versa) and one can rapidly click it to simulate a clock manually. If the simulation is running, the effects propagate immediately – you might see a chain reaction of gate outputs updating as you toggle. If the simulation is paused (user hit space to pause), toggling a switch will still immediately change that node’s state (since that’s a direct user action) but outputs won’t

propagate until they step or run again. The user has control over the tick rate via a slider or input (e.g., 1 Hz up to 60 Hz) [80] . This means they can slow things down to watch carefully or speed it up to see steady-state results.

We align simulation updates with visual updates to avoid jitter: after each tick, the TickEngine notifies the UI to re-render [81] , and we let React or the canvas draw on the next animation frame. This ensures the *appearance* of continuous operation. For example, a clock node might have a little blinking LED; at 1 Hz it blinks slowly, at 10 Hz it blinks faster, up to 60 Hz where it might appear almost solid but flicker (the max tick aligns with frame rate, so it toggles each frame). The use of `requestAnimationFrame` in rendering loops means that even if the simulation tick is slightly offset, the drawing will sync to the display refresh, giving a smooth visual.

Drag & Drop and Selection: Interacting with components on the canvas uses direct manipulation: - Dragging a gate: click and hold a gate, and drag to a new location; the wires attached will stretch or reroute in real-time following the gate (this requires recomputing wire paths as you move things, which we do dynamically). - Selecting multiple: click and drag an empty area to form a selection marquee (lasso). All components (and maybe wires) in that area become selected. Then you can drag the whole selection group to reposition it. This helps in reorganizing a circuit layout for clarity. - Context menus: Right-click (or long-press, or a keyboard context key) on an element might open a context menu with actions (e.g., for a subcircuit chip, context menu might have “Enter/Drill-down” to open it, or “Rotate” if we allowed rotation, etc.). The UI will ensure these menus are navigable via keyboard as well (arrow keys and enter). - Smoothness also means animations: perhaps we animate the appearance of new components (fading or popping in) or animate wires when a connection is made (a quick highlight along the path to show it’s linked). These subtle touches make the interaction satisfying.

HUD and Indicators: The Playground likely has a heads-up display/status bar showing things like simulation status (running or paused), current tick speed (maybe a slider or numeric), and maybe the number of components. If at 60 Hz and the circuit is large, if we ever drop frames or ticks, we might show a warning (FPS drop indicator) – this ties to performance feedback in the UI, possibly an advanced feature for users to know if they’ve hit simulation limits.

Inspector Panel: Clicking a component might bring up an inspector (the code suggests a `PropertyInspector` component [82]). This could allow editing properties (for example, toggling a clock’s frequency or phase, or labeling a node). The interaction here is that the user clicks a component, and a side panel (or modal) shows its details. They can type in a label, choose a color, or adjust a parameter (like the number of bits for a bus or the truth table for a customizable LUT node if any). This ties into exploration – the user can modify how a component behaves and immediately see changes.

Multiple Views & Toggle: The presence of `ViewMode` (circuit, schematic, oscilloscope, 3D) in code [83] suggests the user can toggle between different representations: - *Circuit view* – probably the default live editable canvas with icons and wires (somewhat stylized, possibly pictorial). - *Schematic view* – possibly a more abstract logic diagram (maybe standardized symbols or an auto-generated circuit diagram). - *Oscilloscope view* – a panel showing signal vs time traces for selected nodes (the user can pick which signals to track). - *3D view* – perhaps a fun 3D visualization of the circuit (like a floating chip or using Three.js to render the circuit in a pseudo-physical layout). The user can click toolbar buttons or use shortcuts to toggle these views (the code shows maybe they have a split view mode where you can see two at once). This multi-

view approach enriches interaction: e.g., a student can watch the oscilloscope while toggling an input to directly correlate time domain behavior with circuit changes.

We ensure switching views or opening multiple views is smooth – possibly with an animated transition or simply instantaneous but preserving state (e.g., pausing simulation when needed, or continuing it seamlessly). The architecture ensures one source of truth for the circuit, so all views reflect the same state (if a wire is removed in the circuit view, the schematic and scope update accordingly).

Summoning Tools Quickly: We integrated a quick-add palette (`QuickAddPalette` in code) [84] . This likely appears when the user hits a certain key (maybe `Q` or double-tap on background) – a small overlay where the user can type the name of a component and hit enter to place it. For example, the user types "AND" and it highlights the AND gate, then pressing Enter places it under the cursor. This satisfies experienced users who prefer not to drag from the toolbar. It's analogous to the command palette but specifically for adding components.

Preventing Mistakes and Providing Recovery: Interaction design also accounts for mis-steps: - If a user tries to connect two outputs together, or create a short, the system either disallows it or flags it without crashing. Possibly, we could allow it but show a warning icon on the wire meaning "this is a conflict" so the user learns from it, rather than forbidding – it depends on educational intent. Initially, we might simply prevent it for simplicity. - If a user deletes something by accident, Undo (`Ctrl+Z`) is their immediate recovery. The UI might also have a subtle "undo" toast ("Gate deleted – Undo?") to clue in beginners. - If the user makes a wiring error that causes a logical oscillation or a combinatorial loop with no delay, the simulation might detect it. Because our engine requires explicit `Delay` nodes for feedback loops to stabilize latches [54] [55] , if the user creates a loop without a delay, what happens? Possibly the engine will not settle (it might oscillate or consider it an invalid circuit). The UI can catch this and pop up a non-intrusive warning: "Feedback loop detected – add a Delay component to resolve storage loops." This is part of user education built into interaction (leading them to correct design). - The entire UI is non-modal except when necessary (like Save dialogs or confirmation prompts). That is, the user can do things in any order – they could wire while simulation runs, they can open an inspector while the circuit is still active, etc. Modal interactions (like the Save As dialog) are used sparingly and consistently (with overlay backdrop etc.) [85] [86] .

In summary, the interaction model strives to make the user feel in control of a live digital workbench. The combination of a **keyboard-empowered workflow** (for speed and precision) with **intuitive mouse interactions** (for visual tasks like drawing wires) gives a "best of both" experience. By eliminating unnecessary mode switches and keeping the simulation live, we shorten the edit-run feedback loop to effectively zero. And by smoothing all common actions (wiring, dragging, toggling, menu navigation) to be 60fps and reactive, the tool feels less like software and more like an extension of the user's thought process – they think it, they do it immediately. This immediacy and fluidity are crucial for maintaining flow, which in turn is crucial for learning and creativity.

Performance Model (rAF Alignment, Sim/UI Priorities, FPS Budgets)

Performance is a critical non-functional aspect of Logic Playground: a sluggish or choppy interface would break the illusion of a "live" circuit and could frustrate users, whereas a smoothly running system fades into the background and lets the learning take center stage. Our performance model is designed to keep the

app **responsive at 60 frames per second** under typical use, and to degrade gracefully when circuits become very large or complex.

Simulation Loop and Rendering Synchronization: We use a **tick-based simulation** decoupled from rendering, as decided in ADR-0002 [87] [88]. The simulation (`CircuitEngine.tick()`) advances the logic state in discrete steps, and a `TickEngine` scheduler triggers these ticks at a configured frequency (Hz) [89] [90]. After each tick, the TickEngine notifies the UI to update [91] [81]. In practice, this means after computing a tick (which might take a few milliseconds if the circuit is big), we call `setState` or otherwise signal React to re-render the changed outputs. React (or our canvas draw) will batch these updates and paint them on the next `requestAnimationFrame`. We effectively achieve *rAF alignment* because we don't attempt to render faster than the screen can update (even if the tick is 60 Hz, that at most matches typical display refresh). On slower tick rates (e.g. 10 Hz), the UI still only updates on those ticks. On faster user interactions (e.g., dragging a wire), those are decoupled from simulation and handled directly with rAF – for example, as the user drags, we draw at vsync rates; the simulation in the background might still be ticking but its updates are minor relative to drawing an active drag.

We explicitly rejected tying simulation to rAF (the continuous simulation approach) because we need control over simulation speed (for slow motion and pausing) [92] [93]. But we ensure that when simulation is at max speed (60 Hz), it doesn't conflict with rendering – essentially each tick corresponds to a frame, which is okay. If simulation were to exceed display frame rate (which we don't allow by design, max 60 Hz), it would just cause multiple ticks per frame and we'd only render the latest state on the next frame, effectively dropping intermediate frames (which is acceptable). In testing, we found 60 Hz simulation with small circuits is fine (a tick takes microseconds to a millisecond for tens of gates). For larger circuits, we might not sustain 60 ticks per second, which leads to the next point.

Simulation vs UI Priority: The app's main thread does both simulation and UI (we opted not to use Web Workers in v1 to avoid complexity [94] [95]). Therefore, we must manage the workload to maintain UI responsiveness: - **User input always takes precedence.** We design our event handlers such that user actions (mouse moves, key presses) are processed immediately by the browser event loop even if a simulation tick is ongoing. In practice, since each tick is a loop over nodes, if the circuit is extremely large and a tick computation runs long, it could in theory block input. Our approach to mitigate this: - Keep circuits within manageable size or complexity per tick for v1. Based on our research and tests, circuits up to a few hundred nodes can tick well under 16ms on modern hardware [96]. That is our target for sustaining 60 FPS. - If circuits grow larger (500+ nodes), the tick might take longer than 16ms, meaning we can't do 60 Hz simulation; the user might have set it slower anyway. If not, we drop tick rate dynamically: essentially the TickEngine can't keep up, so the real achieved tick rate falls below 60. That's okay – the simulation slows down but the UI can still render each tick and user input is still handled between ticks. - We ensure not to queue multiple ticks concurrently; i.e. if a tick takes 30ms and the user asked for 60Hz (16ms interval), we effectively run at ~33Hz because we only start the next tick after the previous finished (the setInterval approach might cause piling up if not careful, but we'd adjust via not letting multiple overlaps). - **UI updates are consolidated.** We make sure that state changes from simulation are batch-applied. For example, if in one tick 10 different nodes change output, we don't re-render 10 separate times – we update the central circuit state and then let React reconcile once. This is inherent in how we use state: we likely have a single state object representing all node outputs or the entire circuit. We either replace it or mark changes and have one state update event per tick.

Frame Rate Budgets: We set specific performance budgets: - **Boot-up:** The OS boots in <3 seconds on a typical device [5]. This includes loading all packages, showing the BootScreen animation (which is actually a fixed ~15s sequence by design, but it's non-blocking in the sense of interactivity after it completes). The budget here is mostly about perceived performance – we give something interesting (the boot animation) immediately so the user isn't staring at a blank screen. - **UI Responsiveness:** At all times, the UI should respond to input within 50 milliseconds (way under the human noticeable threshold for UI lag ~100ms). This means if the user clicks a button or drags a component, the system should begin the corresponding action on next frame. Our use of rAF for animations ensures, for example, when dragging, the position updates the same frame as the mouse event or by next frame. We avoid doing heavy calculations in event handlers. For instance, on a drag move event, we simply update element position state and defer any expensive recomputation (like recalculating connection paths) to either a worker or a future tick. However, wire path recalculation might be done on the fly – but even that we optimize by using simple heuristics (straight line to target while dragging, then finalize path on drop). - **Simulation throughput:** We aim for small-to-medium circuits (tens of gates up to a couple hundred) to run at 60Hz without frame drops [97] [96]. Our tests show the O(N) tick algorithm can handle ~100 nodes easily within 16ms on typical hardware (assuming a basic gate evaluation is very fast). At 500 nodes, we might be near the edge on slower machines; in such cases, the user can always lower the tick rate (which is expected for complex circuits – e.g., one might run a CPU at slower Hz to watch it step). The UI budget is thus adaptive: *maintain 60 FPS UI even if simulation slows.* In practice, if a tick takes 30ms, we might only manage ~30FPS rendering, but as long as the UI thread isn't locked to the point of dropping interactions, it will still feel okay (like a game dropping to 30fps – not ideal, but tolerable). - **Remedying Overload:** If a circuit is extremely large (1000+ nodes), our design assumption is that's beyond v1's typical use (we don't expect novices to wire 1000 gates manually; that's more for a computer-built-by-computer scenario which is layer 8, a far-end goal). In such cases, we might provide feedback: e.g. if simulation frame rate drops significantly (detected by measuring tick interval vs target interval), we could automatically reduce the tick rate or pause and show a notice "Circuit is very large; consider pausing or simplifying for better performance." However, the user can always manually reduce the tick speed if they just want to see eventual outcomes. - We deliberately left out complex optimizations like event-driven simulation or multi-threading for v1 because of our target scale and need for determinism/simplicity [98] [94]. Event-driven could be more efficient for sparse activity but introduces complexity in ordering and debugging, and threads (Web Workers) add overhead and race conditions which conflict with determinism (unless carefully synchronized). Instead, by v2, if needed, we might consider compiling the circuit to WebAssembly or using worker threads purely for heavy lifting while maintaining determinism by design. For now, we accept the linear tick cost.

UI Rendering Performance: We use Canvas/WebGL for the circuit rendering (`rb-logic-view`). This means drawing gates and wires is done with efficient graphics operations, not individual DOM elements for each gate/wire (which would be slow for large circuits). The `LogicCanvas` likely handles drawing in one or few canvases. We batch draw elements each frame, and only redraw the parts that changed (though a naive approach might just redraw everything each tick, which is acceptable up to a point given canvas can draw hundreds of shapes quickly). If performance becomes an issue with many components on screen, we could optimize by layering canvases (static background grid in one, components in another, dynamic highlights in another, etc., so we don't clear/redraw everything always). Right now, since our target is a few hundred elements, full redraw at 60fps is fine.

Zooming and panning the canvas are optimized by using canvas transforms or WebGL camera, so we're not re-laying out DOM. This ensures that even when a user scrolls or zooms the circuit, it remains a GPU operation (if WebGL) or a quick canvas redraw, which is fluid.

Prioritization of Tasks: We ensure that heavy non-UI tasks (like saving to file, or complex pattern-matching in a circuit) are done asynchronously or in idle time. For example, the pattern recognition algorithm checks the circuit structure for known patterns [99]. If that were done on every edit synchronously, it could slow down editing. We implement it with debouncing and possibly run it in a web worker. Indeed, the summary says pattern recognition is debounced ~2s after last edit [100], so it doesn't stall the UI while user is rapidly wiring. By delaying it until the user pauses, we utilize idle time for analysis, thereby not interfering with immediate interactions.

FPS Monitoring: During development (and possibly as a hidden feature in production), we include a performance overlay or console logging of simulation and render frame rates. This helps catch any regression – for example, if adding a new feature inadvertently slows the tick loop, we'll detect that the sim can't hit 60Hz on a baseline test circuit and address it. In CI or definition of done, we might include a test of performance: e.g., simulate a 100-node random circuit for 100 ticks and assert it completes within (100 * 16ms * some factor) to ensure performance hasn't degraded (this is tricky to automate across environments, but we can have baseline benchmarks).

Memory management: While FPS is one aspect, we also keep an eye on memory usage. The circuits are not huge data-wise, but if a user builds very complex circuits, thousands of nodes might consume a few MB in memory (which is fine). We do need to be careful with the history stack memory: storing 50 states of a large circuit means 50x memory. If each circuit state is, say, 1MB (which would correspond to tens of thousands of elements), 50MB might be borderline but probably okay in modern desktops. Typically circuits will be smaller. If needed, we could compress history by storing diffs instead of full snapshots, but that complicates undo logic and could risk determinism if not done carefully. For now, full snapshots are simplest and given our scale, acceptable (50 * maybe at most 100KB for a few hundred nodes circuit = 5MB, trivial for desktops).

Rendering Budgets: We expect typical circuits (like a 4-bit adder) to have maybe 20 components and 30 wires – trivial to render at hundreds of FPS. A more advanced one like a simple CPU might have a few hundred components and wires – still within the range that canvas can redraw at 60fps. If someone tries a 8-bit CPU with thousands of transistors (like not abstracting anything), they might push limits. But by then, they should be using chips to abstract repeated structures which cuts down render complexity (each chip drawn as one icon instead of many gates internally). So our very design of encouraging hierarchical chips also serves performance by reducing on-screen element count for large designs. In effect, the *educational progression aligns with performance optimization*: as designs get complex, users naturally encapsulate them, which keeps the visual complexity (and thus render cost) manageable.

Animations and requestAnimationFrame (rAF): We align all custom animations (like the BootScreen progress, the UniverseOrb rotation glow) to rAF as well, using CSS or JS with rAF for timing. This ensures those animations are smooth. The BootScreen, for example, is 15 seconds with presumably CSS animations for the progress bar and orb – these run at vsync for fluid motion [101]. During boot, nothing else heavy is happening, so 60fps is easy. In-app animations such as window minimization or chip modal opening use CSS transitions or a common animation library with rAF under the hood. The window manager might animate windows when moving or switching, but those are lightweight (just transform CSS likely) and don't conflict with simulation because usually you're not dragging a window while the sim is churning heavy (and if so, the sim cost is still minor relative to render).

Future Hooks: If performance ever became an issue, we have planned points to improve it: - The architecture allows plugging in a WebWorker to run ticks in parallel if needed (not trivial due to data transfer overhead, but possible for large circuits). - The `rb-logic-adapter` suggests there might be utilities to sync only changed values to the view, so we don't re-evaluate unaffected parts of the circuit unnecessarily. - If we implement advanced circuits with propagation delay simulation (like analog timing), that would complicate performance; but we explicitly keep it digital discrete for now to keep performance predictable and high.

In conclusion, our performance model ensures that under expected workloads, the system **feels realtime and responsive**, hitting 60 FPS in UI updates and maintaining simulation determinism without dropping frames. Through careful scheduling (tick vs render), controlling complexity, and utilizing efficient rendering (Canvas/WebGL), we achieve a snappy experience. And importantly, by monitoring and adhering to budgets (like not exceeding certain node counts per tick without user noticing slowdown), we avoid nasty surprises. If the user pushes beyond, the system fails gracefully: the simulation simply runs slower but the UI remains interactive, preserving the user's ability to correct or inspect what's happening. This way, performance issues never halt the learning process; at worst, they become another observable aspect (e.g., "wow, my very complex circuit is running slow, just like a bigger program might on a slow CPU – perhaps I need to optimize or break it down," which in itself is a learning moment we could embrace in advanced lessons on efficiency).

Testing Strategy (Simulation Determinism, UI Regression, Perf Tests)

We adopt a rigorous testing strategy to ensure that RedByte OS and Logic Playground function correctly today and continue to do so as new features are added. Our tests target the logic simulation's determinism, the correctness of user-interface behavior (including edge cases), and the performance characteristics over time.

Unit Testing the Logic Engine: The `rb-logic-core` is the most critical piece to be absolutely correct and deterministic. We have a battery of unit tests for the LogicEngine and Node behaviors [102]. For example: - **Truth Table Tests:** For each basic gate (AND, OR, NOT, etc.), we verify its output for all input combinations. These are straightforward but fundamental. - **Propagation Tests:** We set up small circuits (like two gates chained, or a fan-out from one input to multiple gates) to verify that one tick of the engine accurately propagates signals to all affected outputs. We test scenarios like a NOT feeding an AND – flip the input and after one tick, the AND's output reflects the NOT's inversion, etc. - **Sequential Logic Tests:** We include tests for flip-flops and latch behavior. For instance, an SR latch built from NOR gates is tested: set S=1,R=0, tick, expect Q=1; then S=0,R=0 (hold), tick, expect Q remains 1, etc. We also test that invalid states (S=1,R=1) produce the expected handling (maybe our simulation leaves outputs as they were or sets an invalid flag). - **Clock and Timing Tests:** The Clock node is deterministic – e.g. if it's supposed to oscillate every tick, we tick the engine 10 times and ensure the clock output alternates 1010... accordingly. If a Delay node is used, we ensure it delays signals exactly one tick (toggle input, tick, output should still reflect old input, tick again, now output updates). - Importantly, we test that the simulation is **pure** and reproducible. We might reset a circuit to initial state, run 50 ticks, record outputs sequence, then reset and run 50 ticks again and assert the sequence is identical [3]. Also, if we pause (stop calling tick), nothing changes – state is stable until the next tick (no hidden timers). This ensures determinism and no side-effects sneaking in.

Integration Tests for Features: We have tests that cover how various pieces work together, often using a headless mode or JSDOM for the React app: - **Undo/Redo Integration Test:** We simulate user actions in sequence and verify state. For example, programmatically add a gate to a circuit (through the same function the UI would call), then add another, then call the historyStore.undo() and check that the second gate is gone but the first remains [18] . Redo and verify it comes back. Also test that the history doesn't drop intermediate states unexpectedly (like do 51 actions and ensure the first one was dropped due to the 50-limit). - **Chip Creation and Usage Test:** As outlined in our Chip System summary, we have tests that simulate the pattern recognition and chip saving process [103] . For example: 1. Construct a known pattern (like an XOR from gates) in the Playground model, call the pattern matcher, verify it returns "XOR" with high confidence [99] . 2. Call the saveChipFromPattern function with that circuit, provide chosen inputs/outputs, and ensure that a ChipDefinition is created, stored, and that the NodeRegistry gets updated with a new composite node type [69] . 3. Then simulate placing that new chip into a new circuit (maybe by constructing a Circuit that includes a node of type "XORChip" or whatever). Run the simulation on that higher-level circuit and verify the behavior matches the original sub-circuit's logic (i.e., the chip is truly equivalent to the gate implementation). 4. Test deleting a chip from the library and confirm it's removed from persistence and no longer available in registry (maybe even test that circuits using it will fallback or error appropriately). - **File Operations and Persistence Test:** Test that saving a file actually puts data in the file store and that opening retrieves the same data. We might use a mock or memory fs for this in tests. For instance, create a circuit, call the save function with filename "test.rblogic", then clear the working memory, call open on that file, and assert the circuit in memory now matches what we saved. Also test versioning: simulate a file with an older version schema and ensure our `deserialize` upgrades it to current internal structure properly (these tests double as migration tests when we bump versions). - **UI Regression Tests:** We employ a combination of automated UI tests using a headless browser and snapshot tests. For example: - **Snapshot Tests:** We render components (like a Gate icon, or the KeyboardShortcutsHelp modal) with known props and take snapshots of the DOM (or a serialized form). These ensure that changes to the UI are intentional. If someone changes the layout or text of the shortcuts help, the snapshot test will flag it and we can review if that's okay. We maintain these for key components like modals, palettes, etc. - **Interaction Tests with Testing Library:** Using something like React Testing Library or Cypress for more end-to-end, we simulate user interactions: e.g., render the Playground, simulate adding a gate via the palette, connect it via dragging, etc. This can be complex to simulate fully (especially the drag & drop), but we can test simpler aspects: clicking toolbar buttons results in expected state changes (e.g., clicking "New" clears the circuit, which we assert by checking the circuit nodes count = 0). - We verify that certain UI flows don't regress. For instance, test that the examples dropdown populates correctly grouped by layer [22] (we can render the dropdown and check that `<optgroup>` labels match "Layer X: description" from our data). - Another example: ensure the tutorial overlay can go through all steps without error – simulate calling `tutorialStore.start()` then stepping through next steps and verifying that at final step the state signals completion. (We might not verify UI visuals in detail here, just that the state sequence is correct and no exceptions). - **Visual Regression / Style Consistency:** We might incorporate a visual regression tool (like Chromatic or Storybook snapshots) to catch changes in CSS that could break consistency. For instance, if a developer inadvertently changes a global style that makes text unreadable (color conflict), our visual snapshots might flag that. However, such pixel-level tests can be flaky, so we might rely more on manual design review and strict adherence to using the design tokens (which we enforce via lint or code review).

Determinism in Tests: We ensure tests themselves are deterministic. Since we emphasize no random timers or asynchronous races, our tests don't have to worry about flaky outcomes. We set fixed seeds if any randomness was used (currently none in sim, but if later we had an AI hint or something that uses randomness, we'd control it in tests). The AI_STATE dev guidelines also mandate deterministic execution and

no flaky assertions in tests [104] [105]. We use utilities like `waitFor` to handle asynchronous UI events in tests, and ensure to clean up after each test (clear localStorage, reset any singleton stores) [106] so tests don't bleed state into each other.

Performance Tests: We incorporate tests and measurements to ensure performance regressions are caught: - **Simulation Performance Test:** We can write a test that programmatically creates a large circuit (e.g., 1000 simple gates chained in a line) and then measure how long 100 ticks take. This can be done in Node environment for speed. While exact timing may vary by environment, we set an upper bound that is generous but will catch gross slowdowns (like ensure 100 ticks of 1000 gates doesn't exceed 200ms on a reference machine). This is a bit tricky to enforce in automated CI due to variability, but we could at least measure complexity class: e.g., double the number of gates and expect roughly double the time, to ensure algorithmic complexity hasn't unexpectedly worsened. - **Memory Leak Test:** We run a scenario: open a circuit, make changes, undo/redo a bunch in a loop (simulate a user editing extensively), close the circuit or navigate away, then force garbage collection (if possible) and check that memory use returns to baseline. Not trivial in automated tests, but we can simulate by monitoring object counts (e.g., if we keep references, ensure event listeners or intervals are cleared). We do have guidelines: e.g., remove event listeners on component unmount (like in Playground code, they remove window event listeners on cleanup) [107]. We might have test hooks to verify no lingering timers (`TickEngine.pause()` should clear intervals [108], we can test that). - **UI Rendering Perf Test:** Using jest or a browser environment, we can mount a component with many children (simulate a big circuit with, say, 500 gates in state) and measure render time or at least ensure it doesn't timeout. Or use browser performance API in an integration test to ensure, say, dragging a node 100px triggers on average <= X calls to heavy functions. These are advanced and maybe not fully automated, but we plan to do periodic manual perf audits (profile the app with dev tools on heavy circuits).

Continuous Integration (CI) and Quality Gates: All tests (unit, integration) run in CI. We treat any failure as a stop-ship. Linting and TypeScript checks are also part of CI – e.g., no TypeScript errors are allowed (strict mode) [109]. We also run formatting checks (Prettier) to ensure consistent code style, indirectly aiding maintainability which affects quality.

User Testing and Feedback Loop: In addition to automated tests, we plan a beta testing phase where real users (or ourselves in dogfooding) try to break the system. We keep an issue tracker for any bugs found and write regression tests for them. For example, if a user discovered that undoing after saving a chip caused a wrong state (hypothetical), we'd write a test replicating those steps and then fix the code. Those tests stay to ensure the bug doesn't recur.

Testing of Non-Goals: We even test that things that are supposed to not exist indeed don't. For instance, since we said "no multiplayer," we ensure no code accidentally tries to sync state externally (one might test that no networking calls are made during normal operation). "No forced account" – ensure that on a fresh start, the app doesn't block any feature behind an auth check (basically, by design it doesn't, so that's inherently satisfied, but in test we could simulate not logged in state and see nothing breaks). While these aren't typical tests, they reflect our commitment to those non-goals: essentially verifying that the system is indeed entirely local by code inspection or test (for example, scanning the codebase to ensure no usage of `fetch` or external URL except maybe some CDN for resources, which we don't have).

Deterministic CI environment: Our CI is set up such that tests run in a deterministic environment (controlled Node/browser versions, headless for UI tests). We use `act()` and appropriate waits to ensure

no test flakiness due to React state updates [104]. If any test is flaky, we treat it as a bug (either in test or code).

Documentation Tests: We have documentation like this spec and accompanying markdown (like KEYBOARD_SHORTCUTS.md, etc.). While not code, we do cross-verify that all shortcuts in the code are documented. Possibly a test can parse the KeyboardShortcutsHelp component data and compare to the Markdown to ensure consistency (or we manually do that check each release).

By employing this multi-layered testing approach – from low-level logic tests to high-level UI and performance checks – we maintain a high confidence in the system's correctness and reliability. Whenever we add a new feature, we will add corresponding tests. For example, if we implement the narrative Marcus pop-ups, we'd add tests to ensure they only appear under the right conditions and can be toggled off. If we integrate a Help App, we'll test that launching a tutorial doesn't interfere with Playground state, etc.

Finally, our Definition of Done (detailed later) explicitly requires all these tests to pass and no new significant decrease in coverage or performance. Only when the test suite is green, with full coverage of new code and no regressions, do we consider a feature complete [109]. This disciplined testing culture ensures that the integrity of RedByte OS remains high as it evolves – crucial for an educational tool that students and teachers need to trust for accuracy and stability.

Roadmap by Phase

The development and rollout of RedByte OS and Logic Playground are planned in progressive **phases**, each with specific goals and deliverables that build upon the previous. Here we outline the roadmap from the current foundation (Phase 0) through future enhancements (Phase 4), aligning features with the educational layers and user experience improvements.

Phase 0: Responsiveness & Coherence (Foundation)

Objective: Make the base system feel alive, polished, and coherent in behavior and appearance.

Phase 0 corresponds to establishing the **Genesis** of RedByte OS – the monorepo structure and core components are set up, basic functionality is in place, and the system is usable albeit minimal. By the end of Phase 0 (which is essentially now, as per the Stage 0 completion report [110]), we achieved: - **Monorepo Initialization:** All core packages and the playground app scaffolded and building [111]. The 15-second animated BootScreen runs on launch [32], giving a sense of weight and fun (the progress bar with rotating tips and the UniverseOrb visualization provide an engaging start). - **Basic OS Shell:** The desktop environment launches, you can open the Playground app in a window, move it around, minimize, close, etc., using the WindowManager. Global shortcuts like Alt+Tab and Ctrl+W for windowing are functional. The OS provides a consistent frame (window chrome, icons, theme). - **Logic Playground MVP:** The Playground at this stage allows placing basic gates and wiring them. You can toggle inputs and see outputs update. Simple examples can be loaded (likely hardcoded ones). The simulation engine works for combinatorial logic, and the UI updates accordingly. The Undo/Redo system is implemented with the 50-step history [112]. It's essentially a single-layer logic editor with fundamental editing actions. - **Coherent Look & Feel:** Thanks to the design tokens and theme, Phase 0 ensured there's no wildly inconsistent UI element. The dark theme with neon/cyan highlights is applied uniformly, icons are all from one set (e.g., consistent style for window control icons and logic gate icons). Typography and spacing follow the token rules. - **Performance**

Baseline: The app is responsive with small circuits. Boot time is within target. No major lag in interactions. Phase 0 might not have stress-tested large circuits yet, but nothing in the architecture precludes scaling.

Phase 0 is essentially **complete** (the "Stage 0 Complete" indicates this foundation is done [113]). The focus here was on *responsiveness* (ensuring a smooth 60fps baseline and low latency in input) and *coherence* (a solid, unified base to grow from). This phase gave us confidence in the architecture and resolved initial integration issues (like setting up build/test tools, migrating any legacy code needed, etc.).

Phase 1: Hierarchical Chip Drill-Down & Breadcrumbs

Objective: Enable creation of hierarchical circuits via reusable "chips" and allow users to navigate into and out of these subcircuits seamlessly.

In Phase 1, we tackle one of the most impactful features: the ability for users to encapsulate parts of their circuit as subcircuits (chips) and build hierarchically. This addresses Layer 2 of the curriculum (reusable chips) and is a direct follow-on to the pattern recognition delivered earlier.

Key deliverables in Phase 1: - **Save as Chip Feature (Completed):** Much of this was implemented in late Phase 0 / early Phase 1 timeframe [114] [115]. Users can take a recognized pattern or a selection of components and save it as a ChipDefinition with a name, description, I/O ports, and an assigned layer (0-6) [115]. This is already functional: the system suggests ports, you confirm, and the chip is stored (persisted) and registered for use immediately [68]. The UI includes a modal dialog for this process [115], with polish like keyboard shortcuts in the modal (Ctrl+Enter to save quickly) and validation of unique chip names [116] [117]. - **Chip Library & Palette:** A new section in the component palette lists saved chips, organized by layer (with color coding perhaps) [68]. Users can browse and choose their custom chips to place. We've implemented a Chip dropdown in the toolbar and a "Place Chip" button workflow [68]. - **Hierarchy Navigation (Drill-down):** This is the big addition in Phase 1. Once chips can be placed in circuits, we need the ability to open them up and edit/view their internals. Phase 1 delivers the **Hierarchy Stack** model (as evidenced by `hierarchyStore` and `HierarchyBreadcrumbs`) [46] [71]. Concretely: - Users can **enter a chip**: e.g., by double-clicking a chip instance on the canvas or via context menu "Edit chip." The UI then switches the view to that chip's internal circuit. Visually, perhaps the window title changes to the chip's name, and the breadcrumb UI appears, showing something like "Circuit > FullAdder" (meaning you went inside FullAdder from the top-level Circuit). - While inside a chip, the user can edit it just like a normal circuit (since it is one). These edits affect the chip definition globally. We ensure determinism: if the chip is used multiple times, editing its definition will reflect in all instances (we update or re-register the component). - Users can **exit** back out: using the breadcrumbs (e.g., clicking "Circuit" to go to top level, or if nested multiple levels, step by step). We also provide an "Exit to parent" and "Exit to top" command in UI (maybe a back button or home icon) [70]. No matter how deep they go, they can always return to the main circuit. - This navigation is designed to feel spatial. Possibly we could animate a zoom-in effect on the chip when entering (but even without fancy animations, the breadcrumb and context change is clear). This addresses the requirement that hierarchical nav feels intentional and spatial. - We maintain the undo history across navigation. That is, edits inside a chip are undoable even after exiting (we likely treat the whole nested editing as part of one timeline, or separate per context but stack them – details aside, from user perspective, they can Ctrl+Z something they did in a subcircuit after coming back out). - **Breadcrumb UI Polish:** The breadcrumb bar shows the nesting path. Each segment is clickable for navigation. It may also have a drop-down if needed (for long names or if many nested, but deep nesting beyond 2-3 is unlikely in v1). - **Testing & Stability:** Ensure that adding/deleting chips and navigating doesn't break simulation. For example, if a user is inside a

chip and hits “simulate,” we likely run simulation only in that scope (or perhaps not allow run inside? But better to allow so they can test a subcircuit by itself). Also ensure if they press run at top level, it naturally simulates through chips (which engine supports via composite node definitions). - **Example Circuits Using Chips:** As a byproduct, we can now include example circuits that demonstrate hierarchy: e.g., an example “4-bit Adder” that uses the previously built “FullAdder” chip. Or a “Flip-Flop” built from latches. These can be preloaded to show off feature.

By the end of Phase 1, the system now fully supports **multi-layer circuit design**. A user could build something like a simple ALU by first building smaller pieces (half adders, full adders, etc.) and structuring them. This unlocks much of Layer 2 and 3 content for the curriculum.

Phase 2: Build Help App & Curriculum Tracks A–C (Education Mode)

Objective: Develop the Help App that provides structured learning content, implementing Tracks A, B, C of the curriculum alongside any needed supporting features.

Phase 2 is where we explicitly focus on the educational content layer. While Phases 0-1 made the platform robust for free exploration, Phase 2 adds the guided learning pathway.

Key components of Phase 2: - **Help App Implementation:** We create a new app module (e.g., `apps/help` or integrated into rb-apps) that serves as the Help/Curriculum App. Its UI likely is a reading pane with interactive elements (text, images, maybe mini-questions). It should be able to launch from the OS (possibly a “Help” icon in the launcher or dock). - **Curriculum Content Tracks:** Define and implement content for: - **Track A: Introduction to Logic.** Covers basics: binary concept, truth values, basic gates and their truth tables, maybe boolean algebra basics. Likely structured as a series of short lessons or a tutorial sequence. Could include small tasks like “wire the lamp to power”, “build an AND gate from two switches and one lamp, and observe output truth table for all combos” [118]. This track targets a beginner (6th grader level): lots of hand-holding and immediate gratification (the first win: turn on a lamp). - **Track B: Intermediate Logic.** Focus on combinational building blocks: constructing XOR from NAND/AND/OR, building half and full adders, understanding how multiplexers work, perhaps introduction of the concept of using binary for numbers (thus building a 2-bit adder). This track is more challenge-based (“Now that you know gates, let’s build something useful out of them”). Emphasis on patterns and using previously learned components. - **Track C: Sequential Logic.** Introduce state: starts with latches and flip-flops, explains the need for clock, builds up to counters and registers, and eventually a simple CPU. This is advanced content (perhaps aimed at high school or college intro level). The track would likely be longer and split into subparts (C1: latches & flip-flops, C2: memory and state machines, C3: building a CPU). - **Interactive Integration:** The Help app content will need to interact with Playground as described. Phase 2 will implement: - The ability for the Help app to load example circuits in Playground via intents [23]. For instance, a lesson might say “Open the example ‘XOR from NANDs’.” The user clicks a button in help, we send an intent to Playground to load that example (Playground likely exposes an intent handler for `openExample(exampleId)`). - Possibly, the Help app could highlight certain UI elements in Playground (though we said no in-canvas overlays; we can instead use text instructions like “Click the AND gate icon in the toolbar”). - Synchronization of steps: The tutorial might wait for the user to complete an action (we can detect in Playground state: e.g., check if a wire exists or a gate output is 1). The Help app can poll Playground via an API or listen to events (maybe via a shared store or messaging). This requires careful design to avoid coupling: an Intent or an observer pattern could do. - **Marcus Narrative Hooks:** Although narrative is Phase 3 in list, some narrative might be integrated here specifically for deep milestones. For

example, finishing Track C building a CPU triggers Marcus commentary. Phase 2 might implement the basics of narrative system so that the curriculum can call it. (Alternatively, narrative is separately tackled in Phase 3). - **Evaluation & Quizzes:** Potentially include self-check questions or mini-quizzes in tracks. For example, after Track A, a short quiz on truth tables or a challenge: "Design a circuit that outputs 1 only when exactly one input is 1" (which is XOR). The app can encourage them to try in Playground and then perhaps show the solution or verify via pattern recognition that they succeeded. - **Needed Supporting Features:** While building the curriculum, we may discover we need extra features: - A **Logic Inspector** that shows truth tables might be helpful early on. Possibly Phase 2 might implement a small truth table viewer that can list outputs for a selected subcircuit. - **Slow-Motion/Step Simulation:** Already in place via tick control [3], but the Help content might lean on it (e.g., instruct the user to slow down to 1Hz and watch a flip-flop). - **Annotations or Highlights:** We might add ability to highlight a wire or gate in Playground from the Help app (like flash it) to draw attention. If so, implementing a highlight mechanism might be done here. - **Track Rollout:** We likely implement Track A first, test it with some users (or ourselves), then B, then C. Track A and B can be completed largely with the features already available by end of Phase 1 (no sequential logic needed except maybe basic concept of a toggle switch). Track C requires that we implement at least a clock and maybe a "propagation delay" or a concept of edge-triggering for flip-flops. In our engine, we do have a Clock and Delay node, but a D Flip-Flop might be built from two latches and a clock gating – we should verify the engine can handle that (the ADR suggests requiring delay for feedback loops [54], which is fine). If we find something missing (like maybe a direct DFF component for convenience), we add it in Phase 2 or early Phase 3 (since Phase 3 is sequential logic in original plan). Actually, per our earlier plan from Circuit Hierarchy phases [119], "Phase 5: Sequential Logic" was originally after chips. But since our Phase 2 demands sequential circuits for track C, we might incorporate Phase 5 (the sequential features) within Phase 2 or just before track C content is delivered. In short, Phase 2 includes adding **Clock node support** (done, we have Clock) and ensuring the engine handles latches (the solution was to require a Delay node to break loops; we might need to provide that to users in track C). The presence of `Delay` node in built-in list [120] indicates it's ready to use for latches (introduce concept of propagation delay artificially). - **User Feedback from Curriculum:** We likely do iterative testing of the tracks with real users or team members and refine. E.g., if track A's flow is too confusing, adjust instructions; if an example is too complex, break it down more.

By end of Phase 2, a newcomer can fire up RedByte OS, open the Help app, and be guided through a comprehensive learning journey: from the simplest logic circuits (Track A) through building meaningful combinational circuits (Track B) to constructing sequential circuits and a simple working CPU (Track C). This essentially fulfills the core educational mission of the project.

Phase 3: UI Grammar & Consistency Polishing

Objective: Refine and unify the user interface across all apps and components, ensuring a seamless and professional UX, and address any usability rough edges discovered so far.

Having implemented most core features by Phase 2, Phase 3 is about going back over the UI/UX with a fine-toothed comb. This includes: - **Consistent UI Components:** Implement the `rb-primitives` library fully, replacing any ad-hoc HTML/CSS in the app with standardized components. E.g., ensure all buttons across the OS use the same `<Button>` component with styles from tokens (some earlier quick implementations might have used raw `<button>` with slightly varying classes – time to unify). All modals (Save modal, Chip modal, Help prompts) should use a common Modal component that handles backdrop, animations, etc. Toast notifications from different parts of the app should look uniform (same color scheme, corner position). - **UI Grammar Audit:** Create a checklist of UI elements: windows, dialogs, forms, palettes, menus,

breadcrumbs, etc., and verify each follows the style guide. For example, check that all headings use the proper font and color (as defined in tokens), all icons align with text properly, spacing is consistent (we use design tokens for margin/padding). - **Interaction Consistency:** Ensure similar interactions behave similarly everywhere. E.g., pressing Escape should close any open dialog or drop-down consistently [121]. Keyboard focus outlines should appear on focused buttons for accessibility, etc. If the Playground uses Ctrl+O for open, maybe the Files app should also respond to Ctrl+O when it's focused (consistency across apps). - **Error States and Edge Cases:** By now, we might have identified certain edge cases – Phase 3 addresses them. For instance: - What if the user tries to delete a component while simulation is running? (We probably allow it; ensure no crash.) - What if they spam undo/redo quickly? (Ensure it doesn't corrupt state.) - Are there any situations where the UI gets into a weird state (like a modal stays open after switching apps)? Fix those. - **Polish & Affordances:** Minor improvements that make the app feel more refined: - Add hover tooltips for toolbar icons (with keyboard shortcut hints). - Maybe animate the docking of windows (minimize/maximize could be animated). - Refine the Dragging visuals (maybe the component you're dragging becomes semi-transparent). - Ensure snapping behavior is intuitive (if we have grid snap, maybe allow holding a key to disable for fine placement). - Possibly implement multi-select box dragging highlight region (if not done earlier). - **Theming & Accessibility:** Phase 3 can also look at theme variations (we had dark neon, carbon, etc. in tokens). Ensure all themes work (no hard-coded color that breaks one theme's contrast). Also consider basic accessibility: high contrast mode if needed, or at least that text is readable (dark mode needed light text, etc. which we did). If time permits, maybe a light theme option for those who prefer it (we have tokens for a light frost theme presumably [30], ensure switching works). - **Cross-Platform Packaging:** UI consistency also means making sure the app behaves nicely in different contexts. Phase 3 might include testing the app in an Electron or Tauri wrapper for desktop packaging. Ensure window controls map properly (the OS windowing inside the app vs. the actual OS native window might need resolution). No major new features, but adjustments like disabling the real OS's Ctrl+W closing the electron window in favor of internal window, etc. - **Documentation & Help Text:** Clean up any placeholder text or missing descriptions. Ensure labels are clear. Update any on-screen keyboard shortcut references if they changed. Possibly integrate a short "Getting Started" in the help app for using the UI itself, not just logic concepts. - **Bug Fixes:** A sweep of all known bugs (tracked on GitHub issues by now) and resolve them. E.g., fix any pattern recognition false positive/negative that users reported, refine threshold if needed. Or fix that one scenario where the simulation didn't reset properly after unloading a circuit, etc.

Phase 3 may not be as glamorous feature-wise, but it elevates the product to a quality level where it feels polished and reliable for all users. By the end of Phase 3, we expect to hit our "UX quality bar" in Definition of Done: no glaring UI inconsistencies, a smooth and intuitive experience even for first-time users, and a distinct visual appeal that looks like a professional application, not an early prototype.

Phase 4: Optional Future Hooks (Accounts, Publishing, etc.)

Objective: Prepare the system for potential cloud-enabled features like user accounts or publishing of content, without affecting offline functionality; implement only if needed, and in a way that does not compromise local-first principles.

Phase 4 is forward-looking and *conditional*. These are not core to the mission, but nice to have for growth or community building: - **Account System (Optional):** If we decide to allow users to log in (for cloud sync or sharing circuits online), Phase 4 would design that system. Likely implementing a minimal **authentication** mechanism (could integrate with a service or use GitHub auth, etc.). However, crucially, this would be optional – the user can ignore it and still use everything offline. If implemented, a user can choose to create

an account, which might enable: - Cloud backup of their circuits and chips (so they don't lose them across devices). - Sync settings across devices. - Perhaps track progress in the curriculum (so a student's progress in the Help app can be saved). - **Publishing & Community Sharing:** Implement a feature for users to publish their circuits or chip designs to a public repository or URL beyond just the share link. This could be: - Integration with a gist or a central gallery where people upload circuits with descriptions. - Or enabling multi-user collaboration (which we said is out-of-scope, and real-time collab remains no-go, but turn-based sharing could happen: e.g., I publish, you load and maybe fork it). - Possibly allow publishing a "project" (multiple circuits, or a curriculum track if someone writes one). - If accounts exist, published items could be tied to user profiles. - **Infrastructure for Cloud:** Under the hood, if we do any cloud, we'd incorporate an API or integrate with a platform. Phase 4 might involve setting up a backend (which is a big step out of the offline world, so this is heavily optional). Alternatively, leveraging existing services (like saving to Google Drive or similar, to avoid building our own server). - **Constraints:** We make sure these features **do not break** the local-first, no-login-needed mode. That means: - The app should not require login to start or to use core features. Perhaps a small "Sign in" button in a corner for those who want. - All cloud operations should be asynchronous and fail-gracefully if offline. e.g., if user hits "Save to Cloud" and there's no internet, it just queues or says "will sync when online" but doesn't impede local save. - No data loss or gating: a user without account can manually share via URLs or files as always; the account just simplifies that or adds redundancy. - **Future Hooks Implementation:** Possibly we just lay groundwork: design the data models such that adding an `ownerId` to circuits is possible, or create placeholders in UI for login. But maybe not fully implement if not needed immediately. - **Mobile/Web Considerations:** Another "future" aspect: we said no mobile-first, but if in far future, we or others want to try it on iPad or so, we should identify what would break (dragging with touch, etc.) and perhaps in Phase 4 note those or do minimal adjustments to not outright prevent it. But building a full responsive UI might be out of scope; still, we could do small steps like ensure UI scales reasonably on different screen sizes (someone might use on a small laptop or tablet with keyboard). - **Extensions and Plugins:** Another optional area: maybe allow advanced users to create custom gate plugins (like define a truth table for a black box node). We touched on this with chips (which is like user-defined components). But maybe future hooking could allow scripting or adding new functionality via a plugin architecture. Phase 4 might consider an architecture for that, if desired, or simply document how one could add a new logic component to the NodeRegistry.

Since Phase 4 is optional, it might not be executed immediately or at all unless there's demand or the product goes public with community engagement requiring accounts.

Summary of Roadmap: By following these phases, we've first built a solid foundation (Phase 0), then empowered advanced circuit design (Phase 1), then layered on the guided learning experience (Phase 2), refined everything for quality (Phase 3), and finally prepared for future expansion (Phase 4). Each phase corresponds to delivering on a key aspect of our mission: Phase 1 delivered on "computers are built from simpler parts," Phase 2 delivered on "teach from first principles through CPU," and Phase 3 ensured the platform is enjoyable and trustworthy to use. Phase 4 sets the stage for scaling out usage (if we want sharing or cloud services later).

Help App Curriculum Design

The Help App provides a structured learning path divided into **three tracks (A, B, C)**, each targeting a different level of logic knowledge. The curriculum is designed such that each track can stand alone for its audience, but together they form a comprehensive course from basic logic to computer architecture. Each

track follows a consistent pedagogical pattern: **Concept → Build → Simulate → Explain → Reflect** , ensuring learners not only do the exercises but also understand and internalize the concepts.

Track A: Introduction to Logic (Gates & Truth Tables)

Target Audience: Absolute beginners (middle school, or anyone with no prior electronics/computer background).

Goals: Establish the fundamentals of binary logic: the idea of TRUE(1)/FALSE(0), how simple gates implement boolean functions, and how these can control outputs.

Content Outline: - **A1. Hello, Circuit!** – Introduces the RedByte Playground interface and the notion of a circuit. First task: *turn on a lamp*. The user is instructed to connect a Power source to a Lamp [122] . This immediate win (the lamp lights up) gives satisfaction. Concept of a "closed circuit" is explained briefly. - **A2. Manual Control (Switches)** – Now replace power with a Switch so the user can control the lamp. They learn how a switch toggles power (concept of 1 vs 0 in a tangible way: 1 = on, 0 = off). Perhaps a quick experiment: "Try turning the lamp on and off with the switch – see how you can control the flow." - **A3. Basic Gates** – Introduce the NOT, AND, OR gates (the core boolean operators) [123] . For each: - Concept: e.g., "AND outputs 1 only if both inputs are 1." Tie it to everyday logic ("AND = both conditions must be true"). - Build: Have the user place an AND gate, connect two switches as inputs, one lamp as output. - Simulate: They toggle input switches in various combinations (the Help app might prompt: "Set A=1, B=0, what is output?"). - Explain: After observing, the app shows the truth table for AND and highlights the combination they just tried [29] . They reflect "Oh, output was only 1 when both A and B were 1." - Repeat similarly for OR, and NOT (NOT is single-input: maybe a lamp that is on turns off when through NOT, etc.). - **A4. Truth Tables & Logic Circuits** – Now that they've seen gates individually, we present the concept of a **truth table** more formally. The user might fill in a blank truth table for a gate to check understanding (small quiz: "Complete the truth table for OR" – with feedback). - **A5. Combining Gates (Intro to Circuits)** – Show that we can wire gates together. Simple example: make an XOR from AND, OR, NOT (this is a bit advanced for track A; maybe instead something simpler like a **NAND** demonstration: build a NAND using an AND followed by NOT to show you can combine). Actually XOR might be Track B material. For Track A, maybe just do NAND and NOR as "not both", "not either" concepts [124] as an optional extension, since NAND/NOR are universal gates. - **A6. Reflection & Real-World** – End of Track A asks the user to reflect on what these gates mean. Possibly a fun narrative: "These logic gates are the building blocks of every digital device. Think about a light that turns on only when two conditions are met (both switches on) – that's an AND gate in action in a real circuit." Marcus might add a quote or anecdote about early computers built from relays (which are basically switches implementing logic).

Emphasis: Track A heavily emphasizes the **immediate connection** between an action and result. It leverages curiosity (what happens if I toggle this?) to drive learning the truth tables. The track uses very concrete examples (lamps lighting, etc.) to ground abstract logic in something visual. Reflection is lightweight – mainly, "Now I know how each basic gate behaves."

By the end of Track A, the learner should be comfortable with the Playground basics (placing components, wiring, toggling, reading outputs) and know the basic logic gates and their behavior. This sets the stage for doing something with them in Track B.

Track B: Intermediate Logic (Muxes & Adders – Combining for Functionality)

Target Audience: Those who know basic gates (from Track A or prior knowledge) and are ready to build more complex combinational circuits. Likely high school students or hobbyists with some interest.

Goals: Demonstrate how combining gates yields useful functions – specifically covering *arithmetic logic* and *selection logic*. Introduce the concept of multi-bit (buses) and the idea that small circuits can solve real problems (like addition).

Content Outline: - **B1. XOR Gate (One or the Other)** – Start with a classic: XOR as "exclusive OR". User is tasked to build an XOR out of gates they know [125] (e.g., XOR = (A OR B) AND (NOT (A AND B)), or other formulation). This might be guided: "We want output 1 if exactly one input is 1. Let's build that step by step." They place an OR, an AND, a NOT, etc., connect as instructed (or figure it out with hints). Simulation: test all combos to verify it matches desired truth table. Achievement: "You built a new gate (XOR) from simpler gates!" Save as chip perhaps. - **B2. Half Adder** – Introduce binary addition for 1-bit. Concept: adding two 1-bit numbers (no carry in) produces a Sum (XOR of inputs) and a Carry (AND of inputs) [126]. The track likely says "Notice those are just the circuits you built: XOR gives sum, AND gives carry." They wire an XOR gate (use the one they built or a provided one) and an AND gate together to a two-bit output (Sum light, Carry light). Simulate: try 0+0, 0+1, etc. See carry only on 1+1. Possibly integrate pattern recognition: after they build it, the system toasts "You built a Half Adder!" which reinforces they've done a known thing. - **B3. Full Adder (Reuse & Relate)** – Expand to adding 3 bits (A, B, Cin). Here highlight hierarchical thinking: They can use two Half Adder chips and an OR gate to make a Full Adder (like common approach). The track guides: "Instead of designing from scratch, take your Half-Adder as a block…" Show wiring of two half adders: one adds A+B, then second adds the sum + Cin, etc., with OR of carry outputs. This could introduce the idea of using chips (if not using actual chip feature, at least conceptually treat them as black boxes). After building, test all 8 combos – help app can supply an interactive truth table or just prompt to test key cases (like 1+1+1 to see the 1 + carry out). If chips feature is live, maybe we actually had them save Half Adder as a chip and now literally drag two instances. That would be a cool demonstration of reuse. It also subtly teaches about hierarchical design. - **B4. Multi-Bit Addition (Ripple Adder)** – Now that one full adder is done, scale up to 4 bits. Introduce the idea of a "bus" or grouping wires (some UI concept like multi-bit wires if supported, or just say "we will use four separate adders, one for each bit"). In Playground, user can duplicate the full adder chip 4 times (or we provide a 4-bit adder example for them to examine if building from scratch is too tedious). The track might simplify: show a 4-bit ripple-carry adder and have the user connect it to inputs and outputs, or fill in one stage. The goal is to illustrate pattern repetition and that now they built something that "does math" – e.g., adding two 4-bit numbers results in a 5-bit result (with carry). They can input, say, 0101 + 0011 and see output 1000 (8 decimal). This is a *wow* moment: from logic gates now we built a mini calculator. - **B5. Multiplexer (Selective Routing)** – Shifting focus from arithmetic to control logic. Introduce a 2-to-1 Multiplexer [127]. Concept: "It's like a railroad switch for signals – based on a selector bit, it outputs one of two inputs." Build: use gates to implement (e.g., use AND gates to mask inputs and OR to combine outputs – standard multiplexer design). They wire it up: Input0 AND (not Sel), Input1 AND Sel, then OR = Output. Test by toggling the select. This teaches controlled flow of signals – important for understanding how ALUs or CPUs choose data. - **B6. Combine: ALU Component?** – Possibly an optional advanced exercise: use a multiplexer to choose between an AND and an OR operation output, creating a tiny ALU that does either AND or OR based on a control bit. This previews how control signals select operations. - **B7. Reflection & Application:** Encourage thinking: "With what you know, you could build a circuit to add any size numbers by scaling up – that's essentially how the addition in a calculator's chip works." Also reflect on

how multiplexers allow decision-making in circuits (point to how computers choose which data to pass, e.g., selecting between two inputs in a CPU).

Emphasis: Track B emphasizes building **useful functions** from gates – we solve concrete problems (adding numbers, selecting signals). It reinforces the power of combining small parts and prepares them for sequential logic by touching on the idea of controlling signals (the multiplexer) and the notion of carrying in addition (which implies sequential concept across bits). The curriculum encourages using previously built components (like reusing half adders) – thereby implicitly teaching modular design.

By end of Track B, learners can construct moderately complex combinational circuits and understand concepts like binary addition and conditional selection. They see practical relevance (half adder → full adder → ripple adder shows scaling, and multiplexer shows how to choose inputs like a simple decision).

Track C: Sequential Logic (FSMs, Registers, CPUs)

Target Audience: Advanced students (e.g., late high school, college freshmen) or self-taught enthusiasts who have mastered combinational logic and are ready for the concept of memory and time in circuits.

Goals: Introduce stateful circuits – ones that "remember" – and show how memory + combinational logic yields finite state machines and ultimately a simple computer. By end, the user builds or at least understands a simple CPU.

Content Outline: - **C1. The Need for Memory:** Motivation – explain there are things logic so far can't do, e.g., "remember a button press" or "toggle output every second". Show a scenario: try to build a circuit that outputs 1 every other time a button is pressed – you can't with just gates, you need memory. This motivates latches. - **C2. SR Latch (Set/Reset)** – Introduce the simplest memory element [128]. Build using cross-coupled NOR gates (or NAND, depending on convention). The track guides: "Connect the output of each gate into the input of the other – this feedback creates storage." This is tricky for a newbie, so heavy guidance given. After building (two NORs), demonstrate: toggling S or R sets or resets the latch, and it holds the state. Emphasize need for careful use (don't set and reset at same time). Possibly mention metastability but likely not here. - **C3. D Latch (Controlled Storage)** – Improve the SR latch by adding a control input (Enable) [129]. Show how gating the inputs (only allow set/reset when Enable=1) yields a D latch (where D input is transparently passed when Enable high, latched when low). Build: maybe provide a chip for D latch or step-by-step with AND gates + SR latch. Simulate: show that with Enable=0, changing D does nothing to output – it's latched. - **C4. D Flip-Flop (Edge-Triggered)** – Conceptual leap: only change on clock edge. Possibly skip deep build (that requires two latches in master-slave). Could explain conceptually or if time, have them connect two D latches in series with opposite clock polarity to make a basic edge-triggered D flip-flop [130]. Might use an provided DFF component to avoid confusion, but ensure they conceptually get that it's like a memory that updates only at discrete times (when a clock goes from 0→1, for instance). - **C5. Registers & Counters:** Now that a flip-flop can store 1 bit, show multi-bit registers [131]. Have user create a 4-bit register (4 DFFs side by side, common clock & enable maybe). Show how it can store a 4-bit number. Build a simple **counter**: connect a register to an adder that adds 1 and feeds back into itself (this requires a feedback loop and a clock). We need the concept of a **clock signal** (the TickEngine's clock node can be used) – demonstrate a 2-bit or 4-bit binary counter circuit (with a Clock input toggling, the register increments) [132]. The Playground simulation here shines: let them run it slow and watch the binary count. - **C6. Finite State Machine (FSM):** Introduce the idea of state diagrams if possible. Perhaps a simple example: a toggle flip-flop (T flip-flop behaves like a JK with J=K=1) – basically it divides frequency by 2 (counter example covered

that). Or a 2-bit Gray code counter. Or even a simple traffic light FSM – might be too large for them to build, but perhaps outline it or give as an example circuit to load and examine. - **C7. Simple CPU (Putting it All Together)** – This is the capstone: integrate combinational and sequential to make a simple stored-program computer [133] [8]. Likely provide a pre-built simple 8-bit CPU as an example (like the classic 4 or 8 instruction simple CPU with accumulator, program counter, etc., as per circuit hierarchy layer 6 [134]). Guide the user through its components: - Memory (ROM) holds instructions. - A Program Counter (just a register that increments or jumps). - An **Accumulator** register and a minimal ALU (maybe capable of add, etc., controlled by opcode). - A Control Unit (hardwired finite state machine or microcode ROM) controlling multiplexers and write enables. This is complex, so approach: show a block diagram, then let them step through an actual program on the provided CPU in Playground. The Help app might have a feature to step the clock and highlight what happens in CPU (like PC value, IR decoding, etc. – if not automated, the text can say "see how the PC incremented, now instruction at address X is in IR"). This consolidates all knowledge: they see memory (registers), control logic (decoders, FSM), and data path (ALU, etc.) in action. Possibly incorporate narrative: Marcus might congratulate them on essentially recreating a fundamental computer concept. - **C8. Reflection:** This is philosophical: “Look what you have achieved – starting from a simple light bulb, you’ve built understanding up to a CPU that can run a (simple) program. Computers are no longer magic – you have seen how each layer works and built many of the pieces yourself.” Encourage them to reflect on any remaining gaps or questions. Possibly pose thought experiments: “What if we wanted to make it run faster? (introduce idea of pipelining or parallelism – maybe hint to advanced content outside this course). Or “What else could you make with these skills? Perhaps a memory unit (RAM), or a different architecture like a 4-bit microcontroller – the sky’s the limit.”

Emphasis: Track C is heavy on **conceptual understanding** because building a full CPU manually is too much, but guided partial builds and interactive exploration drive the point. It emphasizes how state (memory) combined with combinational logic (decoders, ALU) yields complex behavior over time (execution of instructions). The track often uses the simulation’s stepping ability to let users observe temporal behavior (which is new vs. tracks A and B which were static combinational).

Reflection in track C should cement the concept that at a high level, they just learned how a computer can be made of simple parts. They also learned methodology: how to approach big problems by layering (just as we layered tracks).

The curriculum design ensures each track feeds naturally into the next: - Track A gave the building blocks (gates). - Track B used them to make functional units (like adders). - Track C took those functional units and added memory/control to create a system (a computer). This progression is the concept of the 9-layer hierarchy from basic gates to meta-computers [6], distilled into three broad tracks for learning.

Throughout the tracks, the principle of **concept → build → simulate → explain → reflect** is repeated: For instance, in Track B half adder: - Concept: half adder adds two bits. - Build: the circuit using XOR and AND. - Simulate: try inputs. - Explain: discuss why XOR gives sum and AND gives carry. - Reflect: note this is fundamental to binary addition. This pattern ensures active learning.

The Help App will incorporate this flow with text instructions, interactive checkpoints (maybe it waits for the correct output or a constructed circuit, using pattern recognition or known outcomes to verify), and then explanatory text appears, followed by perhaps a quiz or prompt to articulate what happened ("Why did we use an XOR for sum?").

In conclusion, the Help App's curriculum is the guided pathway that complements the open sandbox: it leads users through a narrative of discovery that maps to the increasing complexity features we implemented in the Playground. By the end of Track C, a dedicated learner will have a solid foundational understanding of how logic gates form the basis of computing devices – and they'll have done it with their own hands in the Logic Playground, which greatly reinforces the learning.

Narrative Layer ("Marcus") Design

The narrative layer – personified by **"Marcus"** – is an optional, light-touch storytelling and commentary system overlaying the user's progress. Marcus is envisioned as a wise and encouraging guide (perhaps named after Marcus Aurelius as a nod to wisdom, or just a friendly persona) who occasionally chimes in to provide insight, historical context, or motivational commentary. This layer adds depth and humanizes the learning experience without ever taking control or becoming intrusive.

Key Characteristics of Marcus's Narration: - **Rare and Earned:** Marcus does *not* speak up for every little action. We deliberately keep his interventions *rare*, to make them special and not annoying. They are triggered primarily at **significant milestones** in the user's journey or when the user explicitly seeks a deeper explanation. For example: - The first time the user successfully builds a known important circuit (like when they complete the full adder in Track B or the SR latch in Track C), Marcus might appear with a congratulatory message and a bit of context: *"You've just created a Full Adder – a building block of every computer's arithmetic logic! Back in 1960, engineers had to wire these by hand too."* - When the user finishes a track or a major phase (like constructing the simple CPU), Marcus might reflect: *"From a blinking lamp to a working CPU – take a moment to appreciate how far you've come. This is exactly how early computing pioneers started, piece by piece."* - **Optional and User-Requested:** Marcus can also be summoned via the Help App interface. Perhaps there's a "Marcus says" or a small avatar icon the user can click when they want additional insight on the topic at hand. This way, those hungry for more backstory or analogies can get it, while those who just want to focus on tasks can ignore it. We could also integrate Marcus as a Q&A: the user might ask a question in a help console and Marcus answers (this could be static scripted answers or even an AI integration someday, but initially likely scripted). - **Text-Only, Non-Blocking:** Marcus delivers commentary in text form (maybe in a styled overlay or side panel). No voice acting, no videos – this keeps it subtle. It might appear as a speech bubble or a console message. Crucially, it does not freeze the UI or demand acknowledgement; it can appear like a toast or in a collapsible panel. For instance, a small panel slides in at bottom saying "Marcus: <insightful comment>" and after a while slides away, or sits until dismissed if user wants to read thoroughly. - **Tone and Content:** The tone should be warm, insightful, perhaps a touch witty or historical. Marcus is not there to re-teach the material (the Help App text already does that) but to provide: - **Context:** E.g., when introducing the relay-based memory, Marcus might note "The first computers in the 1940s used circuits very much like your SR latch, except implemented with electromechanical relays!" - **Reassurance:** If the user is in a complex part and perhaps has tried a few times (we detect maybe they struggled to get a circuit right), Marcus might gently encourage: "This part is tricky – even pros sometimes get flip-flops wrong on first try. Keep at it; you'll get it!" - **Connection to Big Picture:** Emphasize how what they just did fits into the broader concept. After a significant build, Marcus might say "What you've built is essentially a tiny ALU – the heart of a CPU. All modern processors have something like this inside." - **Philosophical or motivational nuggets:** After completing the CPU, Marcus could deliver the thesis: "By building this, you've demystified computing. Remember, every complex system is built from simple parts – a lesson that goes beyond circuits." - **Trigger Points:** Implementation wise, we will define specific trigger points for Marcus commentary: - Finishing each track: e.g., after track A, Marcus might share a quote from George Boole or something: "In 1854, George Boole formalized the logic you just learned.

Little did he know it would one day run the world." - Key circuit completions: first time pattern recognition flags a known circuit (like "Half Adder detected"), we could tie a Marcus comment giving historical context or usage ("This half adder is like one 'bit' of the adder circuits in your computer's arithmetic unit. Millions of these operate every second in your CPU."). - Maybe Easter eggs: if a user builds something creative or complex beyond the expected curriculum, we could have a few special Marcus lines. For example, if they somehow build a 3-bit binary multiplier, Marcus could be triggered: "Impressive – you've built a multiplier! That leaps ahead in complexity. Great job exploring beyond the given tasks." - If user tries an explicitly non-goal thing (like tries to find multi-user or network features), Marcus might humorously respond: "Trying to network? I'm flattered, but this universe is offline – for now." This is optional; we likely keep Marcus for educational comments, not error messages (which should be separate and factual). - **Non-Intrusive Implementation:** Marcus could be implemented as part of the Help App or as a small overlay component subscribed to certain stores (like pattern recognition or tutorial progression). It should have a toggle – e.g., in Settings, "Narrative comments [On/Off]" in case someone finds it distracting and wants to turn it off entirely. By default, it's on because it's sparse anyway.

Narrative vs. Tutorial: We differentiate Marcus from the direct instruction of the Help App. The Help App says "Now connect gate A to gate B". Marcus says "Notice how that pattern you created appears often – in fact it's the basis of [some real device]." Marcus never tells the user how to do a task; he reflects on what's been done or why it matters. He is like a mentor leaning over after the student solved a problem, saying "Cool, did you know you just reinvented the wheel? Great job."

Integration with Help Tracks: The content team (us) will have Marcus comments written to align with tracks: - At the end of Track A, a Marcus comment might wrap up: *"Logic is like LEGO – now that you know the pieces, you can start building something amazing. Onward to bigger challenges!"* - In Track B after full adder: *"This is exactly how larger adders are built. In fact, what you made is called a 'ripple carry adder'. It's simple, but as you add more bits it gets slower – this is why real CPUs use tricks to add faster (look up 'carry lookahead' sometime!)."* - In Track C upon building the latch: *"The SR latch you built is essentially what's inside every computer memory cell – though actual memory chips pack billions of those on a tiny chip!"* Possibly we present Marcus as if he's observing their progress, giving these insights.

Persona & Engagement: It's important Marcus doesn't come off as condescending or as a childish mascot. The tone is more mentor or narrator. If we had to analogize, think of Clippy (the Microsoft paperclip) as a negative example of intrusive helper, versus a good documentary narrator who chimes in occasionally to enrich the story. Marcus is the latter.

We might stylize his messages with a little icon (maybe a silhouette of a friendly professor or a robot icon, etc.) so it's clear it's an aside, not core instructions.

Technically: We'll implement a trigger system (the tutorial or pattern recognition can call a function like `Narrative.trigger("fullAdderComplete")`) and the Narrative component will display the corresponding message if not disabled. Possibly slight delay after a trigger so it doesn't pop up at the exact same time as a success toast, etc. We'll queue messages if multiple triggers happen at once (rare).

User Control: If user clicks on Marcus's message, maybe it opens a little panel where earlier narrative messages can be read again (like a log), in case they closed it quickly. Or a link "Tell me more" that might open a web link for deep info (like Wikipedia for half-adder if they're curious). But keep that minimal to not distract if they aren't interested.

In summary, Marcus provides a **narrative layer that enriches** the experience by adding storytelling and context. It's entirely optional and runs in the background of the user's journey, ensuring that those who welcome a bit of story or insight get it, while those who prefer pure hands-on won't be hampered. Done right, it can make the learning journey more memorable – people remember stories and personalities, so having a character like Marcus associate with key learning moments can increase retention and enjoyment.

Non-Goals & No-Go Boundaries

To maintain focus and avoid scope creep, we have explicitly defined certain non-goals and boundaries that we will not cross in this project. These are features or approaches we consciously choose **not** to implement, either because they conflict with our principles or are out of scope for our timeline.

- **No Real-Time Multiplayer:** We will **not** implement real-time collaborative editing or multi-user circuit building in the initial product. The complexity of syncing circuit state across network, handling concurrent edits, and the potential impact on simulation determinism is beyond our scope. RedByte OS is designed as a single-user experience (at least in v1). Users can share circuits via links or files, but two users won't be in the same sandbox simultaneously. This avoids a whole class of issues (conflict resolution, network latency affecting tick sync, etc.). We acknowledge collaborative learning is valuable, so in the future we might consider a turn-based sharing or a spectator mode, but *not live co-editing*.

- **No Mobile-First Design:** We are explicitly **not** optimizing the UI for small touchscreen devices (phones) in this iteration. The interaction model is keyboard-and-mouse heavy (or at least keyboard heavy), and trying to cram the Playground onto a 6-inch screen would compromise usability significantly. Features like drag-and-drop wiring and keyboard shortcuts don't translate well to mobile. While tablets with keyboards might run it (especially if using a browser), we are not treating that as a primary target either. The focus is desktop web (and desktop app via Electron/Tauri). That said, the design is responsive enough for different window sizes on desktop, but not for tiny screens. We won't be building a separate mobile UI or doing extensive touch optimization now. This also means no mobile-specific features like pinch-zoom gestures or long-press context menus will be prioritized. (Non-goal doesn't forbid someone using it on a powerful tablet, but experience may not be ideal.)

- **No Tutorial Overlays in Playground:** We will not employ intrusive on-canvas tutorial arrows, blinking hotspots, or modal walkthroughs in the Playground itself. The decision is to keep the Playground as a free exploration zone. Guidance lives in the separate Help App. So you won't see, for example, the Playground dim out with a big arrow "click here" to place a gate – that style of forced interaction is a non-goal. Any highlighting of UI for tutorial purposes will happen within the Help app context (e.g., a screenshot or a subtle highlight drawn from outside, but not blocking the actual canvas interactions). We want users to feel the sandbox is theirs, not a step-by-step wizard. This boundary respects the exploration vs education split: the user is in charge in the Playground.

- **No Forced Account System:** The app will not require login or account creation to use core features. As per our local-first principle, everything (saving, loading, full functionality) works without any account. Users won't hit a "sign up to save your circuit" wall – that's explicitly forbidden by our design. If we implement accounts (maybe in Phase 4 as optional cloud sync), it will be strictly optional and opt-in. Also, we won't lock educational content behind accounts. Many educational

platforms make you log in to track progress – we choose not to, so anyone can launch and learn immediately. This makes the barrier to entry low, and also respects privacy especially for K-12 usage where getting kids to sign up for an account can be problematic. Additionally, there will be no online dependency – a user will be able to download this as an offline app and use everything without internet or auth.

- **No Cloud Dependency:** Related to no forced accounts – even if internet is available, we are not relying on cloud services for simulation or storage. All simulation is client-side in the browser; all data stored in browser local storage or files. We will not, for example, implement a cloud simulation service or require an online license check. Users have full offline capability. (This is partly implied by local-first, but we state it to exclude ideas like "heavy simulations offloaded to server" – not happening in v1.)

- **No Patching Over Determinism:** We commit to simulation determinism as a strict rule, so we treat any proposal that might break it (like introducing true randomness in logic nodes without controls, or non-deterministic asynchronous events) as out-of-bounds. For example, if someone suggested adding a "analog noise" feature or race conditions as a teaching tool, we would implement it only if it can be deterministic or optional. Essentially, anything that makes results unpredictable by default is a no-go.

- **No Hidden State or Duplication:** This is an internal dev boundary: we won't allow the architecture to have duplicated sources of truth (like two stores both tracking circuit state separately). That leads to bugs. So, we actively avoid that in design and code (this was mentioned in Definition of Done too). This means if a feature seems to require, say, copying the circuit in two places, we find another way (like references or single store) or we don't do it.

- **No Compromising Performance for Minor Features:** We won't add features that we know will significantly degrade performance or responsiveness unless they are absolutely core. For instance, a hypothetical example: adding a real-time 3D physics simulation of electrons moving – cool but likely to kill performance – so no. We prioritize the smooth core loop over gimmicks. Another example: we won't do something like continuous auto-saving every millisecond because that could cause lag – instead do sensible debounce.

- **No Proprietary Lock-in Formats:** We choose not to encrypt or proprietary-lock the `.rblogic` files. They are JSON and we keep them that way. We're not interested in obscuring data to enforce platform lock (contrary to some products). This means savvy users can hand-edit or generate these files if they want. This fosters openness and maybe community sharing. It's a non-goal to "protect" our format – instead we welcome integration (this is in line with being an educational tool rather than a monetized platform in v1).

By clearly stating these non-goals, we ensure that development remains aligned with our core vision and that we set proper expectations for users and stakeholders. If in the future some of these become desirable, we'll approach them very carefully and only if they can be implemented without undermining our fundamental principles (for example, if we ever consider multiplayer, it would only be after v1 is stable and likely as an opt-in experimental feature, not core). For now, these boundaries keep the project focused and on track.

Definition of Done

To ensure that RedByte OS and Logic Playground meet a high quality bar at each release, we establish a strict **Definition of Done**. A feature or milestone is considered "done" only when all the following criteria are met:

- **All CI Checks Passing:** The codebase must pass all continuous integration checks with 0 errors or warnings. This includes:
- **Automated Test Suites:** All unit tests, integration tests, and end-to-end tests must pass 100% [109]. We maintain robust test coverage for new features (we strive for near 100% coverage on critical logic like `rb-logic-core`, and substantial coverage on UI components and flows). If a test is flaky, that's treated as a failure to be fixed – no ignoring intermittent failures.
- **Linting:** The code must pass linting (ESLint) with no errors and ideally no warnings. Our coding standards (style, best practices) are enforced via the linter configuration. This ensures consistency and catches potential bugs (like unused variables, etc.). Phase R of development even sets a gate of "fail on warnings" to keep things clean [135].
- **TypeScript Compilation:** The project compiles with no TypeScript errors (we use strict mode) [109]. Types help catch a ton of issues; any `any` use or bypass should be deliberate and minimal. No `ts-ignore` unless absolutely necessary (and justified).

- **Build Process:** The app bundles successfully for production (via Vite) and development. No broken imports, etc. The output meets our size budgets (the bundle not excessively large beyond expectation).

- **Performance Budgets Met:** The application must meet the performance targets we set:

- **FPS**: The interactive parts (dragging, toggling, simulation of typical circuits) should maintain ~60 FPS on a normal dev machine. We specifically ensure that simulation of a reasonably sized circuit (e.g., 100 gates toggling at 10 Hz) does not drop frames noticeably [97]. If during final testing we see UI jank or sim delays beyond acceptable thresholds, that must be profiled and fixed or documented.
- **Load Time:** The app should load (boot) quickly – we set < 3s as a target for boot screen to finish [5] on a modern browser. If our bundle grew too large and slowed load, we might need to code-split or optimize.
- **Memory**: No obvious memory leaks. Use DevTools to start/stop simulation or open/close windows repeatedly and observe memory – it should stabilize, not continually grow. We specifically test that creating and destroying circuits doesn't accumulate unreleased objects (we ensure stores clear, event listeners detach, etc.).
- **Efficient Repaint**: UI should not be doing excessive repaints or layout thrashing. We test heavy scenarios (like moving multiple components) and ensure no catastrophic slowdowns.

- If any performance test or benchmark we have is failing its threshold, we treat it as not done.

- **Regression Tests and Issue Closure:** All previously reported bugs that were targeted for the release are fixed and verified via tests. We run the full test suite including any new regression tests (if a bug was found and fixed, we add a test for it). Also, a manual regression sweep of core flows (smoke test the major user actions: booting, loading examples, saving, undo/redo, pattern recognition, etc.). There should be no high-priority known bugs open. If some minor issues remain (low priority edge

cases), they are documented – but anything that severely impacts the defined user scenarios must be resolved.

- For example, if during development we noticed a sporadic undo bug and fixed it, we ensure there's a test covering that scenario [103]. If something like "Chip deletion doesn't remove from palette until restart" was a bug and we didn't get to fix, that would usually block done unless truly trivial – ideally fix it or decide it's acceptable (rare case).

- **UX Quality Bar Achieved:** The overall user experience should feel polished and intuitive:

- **UI Consistency:** As addressed in Phase 3, the interface should have consistent visuals and behaviors. We go through a UX checklist: are all buttons labeled or have tooltips? Do icons make sense? Is there any confusing terminology? We often do a UX review or have someone fresh run through it and note if anything felt off.
- **No Dead-End Interactions:** We test that the user can accomplish tasks without getting stuck. For instance, can they always exit a modal with Esc or a cancel button? If they open a file dialog and cancel, app still works fine. Basically, no flows where the app becomes unresponsive or unclear.
- **Error Handling & Messaging:** All user errors (like trying to save over an existing file, or invalid circuit like two outputs shorted) are handled gracefully with an understandable message, not a crash or silent failure. Check logs: no red errors in the browser console during normal use. If any remain (non-critical), fix them or silence if benign but ideally eliminate.
- **Accessibility & Navigation:** While we didn't fully focus on accessibility (like screen readers), we did aim for keyboard navigability and basic focus management. We ensure tab order is logical, focus is visible. It's part of quality bar that e.g., you can open a modal and then Esc to close, etc.
- **Smoothness:** Animations or transitions that exist should be smooth. No jerky or half-implemented animations. If something animates weirdly, either polish or remove the animation.

- Possibly do a small user testing session: get a couple of people who haven’t used it to go through a track or two and see if they encounter any major confusion. If yes, address it (either by UI change or adding help text).

- **Architecture Clarity:** The internal code architecture should be clean and maintainable for future work:

- **No Duplicate State:** As we decided, ensure we aren’t storing the same piece of data in two places causing sync issues. For example, if we at some point had both hierarchyStore.currentCircuit and also a separate global currentCircuit variable, that's bad – ensure only one. We code-reviewed to avoid such situations.
- **Testable Stores:** Key logic in stores (history, chip, file, etc.) should be decoupled enough to test in isolation. Indeed, we have tests for them, meaning they've been written in a modular way (pure functions or small classes rather than deeply tied to React). This is done already (like chipStore is separate and logic-core is pure).
- **No major tech debt or TODOs:** We check the code for "TODO" comments or shortcuts we took. If any relate to critical functionality, they should be resolved or at least tracked. The code doesn’t have hacky solutions that will haunt us unless absolutely necessary and documented. The AI_STATE guidelines also mention tests must not have leftover stuff and no warnings [136], etc.

- **Documentation Updated:** All relevant documentation is up-to-date. That includes in-code docs for public APIs (if someone were to use rb-logic-core), the README for the project, and user-facing docs like the Keyboard Shortcuts reference [109], etc. The circuit hierarchy doc was written, the session summary indicates documentation was updated [109] as features delivered. We should ensure the Help App content covers using features that might not be self-evident (like maybe a note on Undo/Redo in tutorial 0).

- **No state leaks:** Using React dev tools or our own checks to ensure we aren't accidentally keeping state between circuits or duplicating. For example, if you load a new circuit file, the old state should be cleared – test that thoroughly (e.g., if previously pattern recognized something, make sure a new file resets that state).

- **User Acceptance:** (If we had specific user or client criteria) The feature meets the user story acceptance criteria. For instance, if a requirement was "As a user I want to share a circuit via URL", Done means we tested that copying link and opening in a fresh browser indeed reproduces the circuit. All such user-facing requirements enumerated in our initial task list have been verified. In our context, it could mean our own acceptance tests for tracks – e.g., can a new user complete Track A without external help? If we find they can't because instructions were lacking, then track A isn't done – refine it.

- **Ready for Deployment:** We have packaged the app, tried a production build ourselves, maybe deployed to a test server (like a static site or a packaged app installer) and done a sanity check on that environment. Everything works in production mode (sometimes dev mode hides issues that appear when minified, etc., so we double-check).

- Also finalize versioning, update version numbers, changelog (if we maintain one), etc., as part of done.

Only when all above are satisfied do we truly call the release done. It's a high bar, but this ensures that when we deliver the product to users (whether internal or public), it functions as advertised and provides a smooth learning experience. Maintaining this Definition of Done discipline keeps quality high and builds trust: educators and learners using RedByte OS can rely on it working correctly and effectively every time.

Implementation Directive (for Claude)

Scope: This section translates the above specification into a concrete execution plan for the engineering agent (Claude). It breaks down the development into actionable tasks and priorities, ensuring the system is built in the correct sequence. All tasks should be implemented in alignment with the design specified above.

Phase 0: Finalize Responsiveness & Coherence (Foundation) 1. **Verify Monorepo Build & Basic Features:** Ensure the existing monorepo structure is building properly (run `pnpm build` for all packages) and run all tests. Fix any residual CI issues or warnings. The Stage 0 completion indicates this is largely done [110]. Address any minor import/path issues flagged in Stage0 (e.g., fix `rb-apps` import path warnings [137]). 2. **Polish Boot Sequence:** Confirm BootScreen and UniverseOrb components function as intended (15s animated boot) [32]. These are done but test on slower machines to ensure no lag. If needed, allow skip (maybe a hidden key) but not critical now. 3. **Windowing Checks:** Test multiple windows (open Playground twice if allowed, Files app windows). Ensure window controls (minimize, close) work [11].

Implement any missing global shortcuts in shell (Alt+Tab switching, etc.) [13]. If `rb-windowing` lacks something (like restoring z-index on focus), implement per AI_STATE contracts [138]. 4. **Keyboard Shortcuts:** Double-check global and Playground shortcuts mapping to actions [11] [12]. Implement any not done (e.g., Ctrl+Shift+P for command palette if not already, or stub it if palette not implemented yet). Update the KeyboardShortcutsHelp component or markdown if needed to reflect reality. 5. **Basic Save/Load & Examples:** Ensure `useFileSystemStore` and file dialogs are working for .rblogic files. Test saving a circuit, closing app, re-opening and loading it. Also ensure initial examples (if any JSON examples provided) load via appRegistry or `listExamples()` [139]. Fix any file path or persistence quirks. 6. **Performance Baseline:** Profile a simple circuit with continuous tick at 60Hz. Optimize if any obvious bottlenecks (but likely fine). Confirm determinism: run a test of tick output sequence consistency [3]. 7. **UI Coherence Sweep:** Quick visual pass to ensure no glaring inconsistencies in theme (Stage0 claims theme is applied [140]). For now, small misalignments can be deferred to Phase 3, but fix any broken CSS references due to moved files, etc. Confirm dark theme variables load (no bright flash on load). *Outcome:* Phase0 essentially done if tests and basic feature manual tests pass. System feels stable and "alive" (boot anim working, no major lag or placeholder elements).

Phase 1: Implement Hierarchical Chip System & Navigation 8. **Finalize Chip Save & Registry:** The Chip system code from session summary is mostly done [114] [69]. Integrate it fully: - Ensure `saveChipFromPattern(circuit…)` stores chip in chipStore and calls registerAllChips/registerChip [141] [69]. Test saving a pattern (use patternMatcher: build a known pattern like XOR to trigger it). - Implement UI: When pattern recognized (via recognizePattern in Playground), set `recognizedPattern` state [142] which triggers showing SaveChipModal [143]. Ensure SaveChipModal fields pre-fill with pattern name, etc. [115]. Connect its “Save” to actually calling saveChipFromPattern and closing modal. - Chip Library UI: Add a section in ComponentPalette or a separate ChipLibraryModal [144]. Possibly from the UI code: they have ChipLibraryModal imported but usage? Implement a toolbar button "Chip Library" that opens ChipLibraryModal listing all saved chips (from chipStore.getAllChips) grouped by layer [68]. Users can select one and then maybe click "Place Chip". - Place Chip interaction: Possibly similar to selecting a gate from palette. When user selects a chip, set something like `selectedNodeType = chipType` so that clicking canvas places it [145]. Ensure NodeRegistry has the composite node available with proper inputs/outputs. - Test end-to-end: Build Half Adder, pattern recognizes, save as chip, then new circuit, place that chip, simulate to verify it works same as original. 9. **Hierarchy (Drill-down) Navigation:** - Implement `hierarchyStore`: It tracks a `stack` of circuits and `currentCircuit` [46]. Likely already coded. We need to call `enterChip(chipCircuit)` when user wants to edit a chip and `exitToParent()` on breadcrumb up. - UI: Breadcrumb component exists (HierarchyBreadcrumbs) [71]. Integrate it at top of Playground window, visible when stack depth > 1. It should render clickable path (e.g., "Main > HalfAdder"). - Trigger enter/exit: * When user double-clicks a chip node on canvas: detect node.type is a composite (maybe mark composite types). On double-click, call hierarchyStore.enterChip(corresponding Circuit). That likely sets currentCircuit to that chip's internal circuit (which we must store when chip saved). Possibly chipStore can give us a circuit by chipId. * The canvas or rendering logic should now switch to render hierarchyStore.currentCircuit instead of the root circuit. We ensure Playground state uses hierarchyCircuit for its editing target. E.g., in code they have `hierarchyCircuit = useHierarchyStore().currentCircuit` in state [46]. Update all places to use that instead of a single circuit state. * Breadcrumb "Main" click triggers exitToTop (pop to root), "ParentChip" triggers exitToParent (pop one). Implement those to set currentCircuit appropriately [70]. - Ensure undo/redo accounts for hierarchy if needed (maybe we keep one history store but it's probably tied to top-level circuit currently). Could consider separate history per level but maybe out-of-scope for now. Test: do some edits inside chip, undo and see if works within that context. - Mode indicator: Possibly show a distinct background or title

when inside a chip. Perhaps update Playground window title to "Editing: [ChipName]" using useWindowStore.setWindowTitle [146]. - Thorough test: Save a chip, then "Edit" it: double-click chip on canvas, breadcrumb should appear. Modify something inside (e.g., if chip was half adder, flip which gate is output or something), see that it affects instances. Exit and confirm instance behavior changed. - Test weird cases: editing a chip while the main circuit that uses it is running simulation – maybe disallow simulation while in chip edit or auto-pause. - If any issues with chip port mapping or saving (like if user tries to save chip with no IO nodes, chipUtils suggests ports via connection analysis [66]), test those utilities. 10. **Documentation/UX for Chip System:** Update any user-facing tips to mention "you can save circuits as reusable chips". Possibly add a blurb in Help track B or so. Not a code task for Claude, but ensure tooltips or modal descriptions are clear (e.g., SaveChipModal should explain inputs/outputs). 11. **Pattern Library Expansion (optional, if not already):** The Session summary lists recognized patterns [99]. Ensure patternMatcher covers XOR, Half Adder, etc., as per list. If needed, update pattern definitions or threshold. 12. **Finish Phase 1 with Internal QA:** Run through a scenario of building half adder -> saving -> using in full adder -> editing chip -> observe changes propagate. Ensure no crashes. All tests updated for new functionality (e.g., add tests for chip store saving & retrieving, a test for hierarchy store stack push/pop logic).

Phase 2: Develop Help App & Curriculum Tracks A, B, C 13. **Create Help App Structure:** Under `apps/` or as part of `rb-apps`, implement a new React app component e.g., `LogicHelpApp`. Register it in `appRegistry` (with name, icon). This app will contain multiple "tracks" or "lessons". - Design a UI: likely a sidebar or menu to choose track, and a main content area that can display text, images, and have interactive elements (like a "Next" button, maybe embedded small circuits or progress indicators). Possibly use Markdown or hardcode content with JSX. - Basic navigation: allow user to pick a track (A, B, C). Within a track, sequence through lessons. 14. **Implement Track A content:** - Write the content for each step (A1 to A5 etc.) either in code or separate markdown files that we render. Keep it engaging and short paragraphs, use diagrams if needed (maybe ASCII logic gate symbols or embed small image icons from our icons for gates). - Interactive cues: * For example, after explaining AND gate, prompt user "Place an AND gate and connect it to two switches and a lamp. Then toggle inputs." * We need to detect when they've done it. Options: pattern recognition (we could extend it to detect any AND gate with two switches and lamp structure). Simpler: the help app can subscribe to Playground events or query circuit. Possibly use NodeRegistry: if finds an AND with Lamp output, that might suffice. Or just rely on user clicking "Done" if they did it. - Could implement a simple check function for each task. e.g., `lessonCheckers: { stepID: function(circuit) -> bool }`. For AND gate step: returns true if there is at least one AND node present. * For switch toggling test, we might not verify thoroughly if they did try all combos - we might just instruct them and trust they do it. - Build some minimal API: Playground could expose `window.redbyte.getCircuitState()` to the help app for checking, or the help app could directly import relevant stores. - Implement "Next" button that is enabled only when check passes (or always allow skip if user chooses). - Provide a "Skip to solution" or "Load example" for any build tasks if user stuck. - At each concept, include an "Explain" after they simulate: just present analysis text. - Reflection steps: maybe present a question, allow them to think or write (not collecting answer, just rhetorical). 15. **Implement Track B content:** Similarly structure B1..B7. - More emphasis on using chips maybe: by track B we have chip feature, but we might avoid forcing usage unless they've learned it. Might just conceptual reuse. - Possibly include direct integration: after building half adder, instruct them to click "Save as Chip" (since pattern will prompt anyway). Then for full adder, instruct using two half-adder chips from library to assemble it. This elegantly exercises Phase1 features as part of curriculum. Ensure at this point, user has indeed saved half-adder; if not, our text or Marcus might prompt them. - Example tasks like building multiplexer: provide hints, and a "See solution" which maybe loads a prebuilt multiplexer circuit from examples if they get stuck.

- Testing track B flows is important to ensure they can follow without confusion. 16. **Implement Track C content:** The most complex, break into small sub-steps: - Build SR latch: likely instruct step by step (place two NOR gates, connect outputs crosswise, etc.). Possibly supply a partial template (like pre-place gates via an example loaded). - D latch and D flip-flop: we might have them load a pre-made one or walk through less detail. It's heavy, consider using provided chip for flip-flop if time. - Counters: perhaps provide an example and let them experiment toggling clock. Or ask them to connect a few flip-flops linearly to see binary count. - CPU: definitely just load an example CPU (we prepare a JSON circuit for a simple 4-bit CPU, maybe from session summary data). Then the help app guides them to clock through a demo program and highlights what happens (not automatically, but telling them what to observe). - The help content will mostly be explanatory here, given the complexity. Ensure Marcus or help text encourages them and sums up. 17. **Help App to Playground Intents:** Implement linking: - When help says "Open Example X", provide a button that calls `appRegistry.getApp("LogicPlayground").openExample("exampleId")`. If not available, at least copy to clipboard or instruct manual. - Possibly integrate with intents properly: define an intent object with type (like "openExample" and payload). The OS shell can route it to Playground app instance. - Simpler hack: the Help app can directly call a function exposed via window or context in Playground (since it's same environment). - E.g., Playground could expose a global function to load example or do an action, and the help content triggers it. We'll implement whichever easiest for now. - Key is enabling some automation: e.g., at CPU step, have a "Load CPU Circuit" button. - Another: at half-adder step, after building, might have "Check circuit" that uses pattern recognition or scanning to confirm it's correct (like pattern recognizer should identify it as half adder if built right). 18. **UI of Help App:** Make it user-friendly: - Navigation: allow going back to previous steps, or skipping ahead if desired (maybe not skip tests easily, but track selection or step selection might be okay for returning users). - Progress: maybe a progress bar or simply "Step X of Y" labels. - Keep text segments short with bullet points or images as needed (we can reuse some ASCII diagrams from our doc). - Test on various screen sizes (but since desktop, assume window large enough). - Ensure it doesn't hog too much CPU (just text). 19. **Testing & Tweaking Tracks:** Thoroughly run through each track as if a student: - Ideally have a non-developer try it and see if instructions are clear. - Ensure no step is impossible or too ambiguous. Adjust phrasing or add hints if needed. - Check that any checks for step completion work reliably (not false negatives). - Confirm that interacting with help doesn’t break Playground state (like help loading an example doesn’t wipe their unsaved work without warning). - Clean up any console logs or development artifacts in help app. 20. **Integration of Marcus Narrative:** Use triggers in help flow: - E.g., after user builds full adder, trigger Marcus with context text. - Possibly tie into pattern recognition or track completions for narrative triggers. - Implement narrative display: a small overlay at bottom or in help app UI. Possibly simplest: within Help app content, include Marcus quotes at pre-planned points (so it’s part of content rather than truly separate triggers). - If doing separate overlay, implement a Narrative component and call it at triggers. - Ensure toggle in settings to disable narrative if needed. - Use the pre-written lines from spec as guidance for what to say at triggers. 21. **Final Documentation and QC:** Update README or user guide to mention Help App usage, etc. Ensure the application icon for Help (maybe a book icon) is added and everything has proper tooltips.

Phase 3: UI Grammar & Consistency 22. **UI Component Consolidation:** Review all UI elements across OS and Playground: - Replace any raw HTML repetitive structures with shared components from `rb-primitives` if available. E.g., ensure all modals use a common `<Modal>` styling (Stage0 might not have that, we create a basic one). - Ensure `rb-tokens` values are used (no hard-coded hex colors in CSS, use theme variables). - Standardize buttons: same className or component so they have consistent hover effects, border-radius, etc. - Check fonts and sizes for headings, normal text, code (maybe differentiate monospaced font for circuit node labels?). - Align icons with text properly (vertical center, etc.). - Verify dark theme readability: e.g., if some text is gray on dark, ensure contrast >= AA. - Clean up any leftover inline

styles or Tailwind if not consistent (we may be using Tailwind classes given presence of it in Playground tailwind.config; ensure tailwind usage is consistent with design tokens). 23. **Focus & Keyboard Navigation:** Test tabbing through interactive elements: window controls, file list, help buttons, etc. - If any components aren't focusable that should be (like the canvas might trap tab?), handle accordingly (maybe skip canvas in tab order). - Ensure pressing Enter on focused buttons triggers them, etc. - Add `aria-label`s or `title` attributes to icons without text for accessibility/tooltips (like window minimize icon). 24. **Error/Edge-case Handling:** Induce common errors to see user-facing result: - Try to open a malformed file (we should handle or show "Invalid file" modal). - If pattern recognition didn't find anything, ensure it silently does nothing (no error). - Save as chip with no selection (we probably only show SaveChip after pattern, but if user manually triggers chip save someday, handle it). - If user uses unsupported browser or tries to use in incognito (localStorage issues?), ensure degrade gracefully (maybe just note that saving won't persist). - Any console errors encountered in unusual sequences should be fixed or caught. 25. **Polish Visual Details:** - Animations: e.g., add a slight ease-in-out to window moving/resizing if trivial, to make it feel smoother (optional). - Make sure the Orb animation looks good on all device pixel ratios (no weird pixelation). - Check that scaling the browser zoom doesn't break layout (e.g., breadcumb overflow). - Check on different OS/browsers for font differences (just to ensure nothing critical). - Possibly incorporate a light theme toggle to see if all elements adapt (if we have a light theme in tokens). 26. **Memory & Performance Quick Sweep:** After all feature additions, run performance tests again: - Ensure new Help App doesn't drastically degrade performance when open (shouldn't, it's mostly static content). - Check memory after completing a full track with multiple example loads – ensure no substantial leaks (especially event listeners from Help triggers are removed if help app closed). - The increased complexity (chips, etc.) still runs smooth in typical usage. 27. **Bug Fixes from Testing:** Fix any minor bugs logged during track usage or QA. For example, if undo in a chip context had an issue, resolve it now. - If any non-goal by design thing is reported (like "can't do X on mobile"), that's acceptable as known. 28. **UI Final Review:** Run through entire user flow as a new user: - Boot, open Playground, maybe play randomly, then decide to follow tutorials, complete them, then try saving work, share via link, etc. - Everything should feel coherent and free of obvious flaws. If something still feels off (like modal position, or missing feedback on button click), add finishing touches (maybe a click sound? probably out-of-scope). - Confirm that the app icon and name appear correctly if using an Electron wrapper (just in case we pack it). 29. **Documentation & Prep for Release:** - Update any user guide or QuickStart doc to reflect final UI (screenshots if needed). - Prepare a changelog or summary of features for release notes (especially if others will see this). - Ensure version bumped appropriately in package.json if versioning. - Clean up repository: remove any dev-only console.logs, ensure tests are all enabled (no .skip left).

Phase 4: (Optional, skip if not immediate) Hooks for Accounts/Publishing – *If skipping, note as future tasks.* 30. **Scaffold Account System (if decided):** (Optional) Prepare backend integration points: - maybe create a stub Auth module that can later be expanded. Possibly integrate a simple login UI that is hidden by a feature flag. - Confirm nothing in core design blocks adding this later (we designed offline-first, so we're fine). - If time permits, maybe implement an "Export to cloud gist" that posts circuit JSON to a gist or paste service, to simulate publishing (not truly accounts, but a one-click share alternative). - These tasks are low priority and only if resources/time remain after core. 31. **Final Q/A and Approvals:** Confirm that all acceptance criteria from problem statement are satisfied. For example, ensure "keyboard-first everywhere" – we test a scenario with only keyboard and it is possible to do major actions (open help via shortcuts, navigate, etc.). Also ensure local-offline scenario fully working (maybe disable network and try everything). - If any feature doesn't meet spec, fix or adjust expectation if truly minor.

Finally, with everything above done, package the application (for web: build to `dist/`, for desktop: if using Electron or similar, bundle it) and perform a final sanity test on the packaged version.

Deployment: Deploy to the target environment (e.g., GitHub Pages for web demo, or produce an installer for offline use). Confirm one more time in deployed form.

After these steps, the system should be ready for the end-users with confidence in quality and fidelity to the spec. Each numbered task above can be executed in order (some in parallel if multiple engineers), but roughly following the phases ensures dependencies (e.g., chip system needed before help content that uses it) are met.

Claude, please proceed with implementing these tasks, starting with Phase 0 foundations and moving through Phase 1 and Phase 2, while adhering to the design guidelines and global contracts outlined in the specification. Let's build RedByte OS to meet its vision step by step. 147 148

---

1 8 119 122 123 124 125 126 127 128 129 130 131 132 133 134 CIRCUIT_HIERARCHY.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/CIRCUIT_HIERARCHY.md

2 5 9 10 20 31 41 42 43 50 51 52 120 148 genesis.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/docs/specs/genesis.md

3 4 53 54 55 80 81 87 88 89 90 91 92 93 94 95 96 97 98 108 adr-0002-logic-engine-ticks.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/docs/specs/adrs/adr-0002-logic-engine-ticks.md

6 7 18 19 22 61 99 100 109 112 118 139 SESSION_SUMMARY.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/SESSION_SUMMARY.md

11 12 13 72 73 74 75 KEYBOARD_SHORTCUTS.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/KEYBOARD_SHORTCUTS.md

14 15 16 21 25 26 27 28 30 32 101 102 110 111 113 137 140 147 STAGE0_COMPLETE.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/STAGE0_COMPLETE.md

17 23 24 36 37 38 62 63 104 105 106 135 136 138 AI_STATE.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/AI_STATE.md

29 README.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/packages/rb-logic-view/README.md

33 34 35 39 40 46 47 48 49 59 60 64 70 71 82 83 84 107 142 143 144 145 146 LogicPlaygroundApp.tsx
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/packages/rb-apps/src/apps/LogicPlaygroundApp.tsx

44 45 56 57 58 65 66 67 68 69 103 114 115 116 117 141 CHIP_SYSTEM_SUMMARY.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/CHIP_SYSTEM_SUMMARY.md

76 77 78 79 85 86 121 KeyboardShortcutsHelp.tsx

https://github.com/swaggyp52/redbyte-ui-genesis/blob/1371075957b568b221e4ad4375d6fa70e4a03cfd/packages/rb-apps/src/components/KeyboardShortcutsHelp.tsx