Intent-Driven OS

Frontend Architecture Specification

Next.js Implementation Guide

Generated: February 06, 2026

Executive Summary

This document outlines the complete frontend architecture for the Intent-Driven OS, built on Next.js with a hybrid server-client component model. The architecture supports 12 core applications, 3 system-level modes, and enables over 70 unique workspace configurations through intelligent composition.

Key architectural decisions include browser-based file storage using IndexedDB, Zustand for state management, and a modular app registry system that enables dynamic workspace composition based on natural language intent.

Table of Contents

1. Workspace Architecture

1.1 Core Concept

A workspace represents a complete layout configuration that defines what applications are visible, where they are positioned, which system modes are active, and how they are arranged. Conceptually, a workspace serves as a recipe that the rendering engine follows to construct the user interface.

1.2 Workspace Components

Component	Description	Example
Applications	List of active app instances	Notes, Timer, Todo
Positions	Pixel coordinates for each app	x: 0, y: 0, width: 400
Modes	Active system-level modes	focus, dark
Layout Strategy	Arrangement algorithm	grid, floating, split

1.3 Three-Layer Architectural Model

Layer 1: Workspace Manager

The top-level orchestrator that maintains the current workspace configuration, receives parsed intent from the server API, instructs the rendering system on what to display, and maintains a history of the last ten workspace sessions for quick recall.

Layer 2: Layout Engine

The middle layer responsible for receiving workspace configurations, calculating actual pixel positions based on screen dimensions, implementing different layout strategies such as grid, split, or floating arrangements, and managing z-index ordering for overlapping windows.

Layer 3: App Renderer

The bottom layer that receives rendering instructions specifying which applications to display at which coordinates, instantiates the correct component for each application, passes down active mode information, and handles application-specific state management.

2. Application Architecture

2.1 App Registry Pattern

The application system utilizes a central registry pattern that maps application identifiers to their corresponding components, stores metadata including default sizes, supported modes, and purposes, and provides the layout engine with the information needed to instantiate applications dynamically.

2.2 Individual Application Structure

Each application is designed as a self-contained unit with clearly defined boundaries and responsibilities. Applications operate independently while conforming to a standard interface that enables workspace-level orchestration.

Component	Purpose
Props Interface	Receives position coordinates, dimensions, active mode array, and app-specific configuration
Internal State	Manages application-specific data such as note content, timer countdown, or task lists
File System API	Interacts with the unified file system through standardized read and write operations
Mode Awareness	Adjusts user interface based on active system modes, such as hiding toolbars in focus mode

2.3 Application Communication Model

Applications do not communicate directly with each other to maintain loose coupling and architectural clarity. Instead, they utilize an event-based communication pattern where applications publish events to the workspace state manager, other applications subscribe to relevant events, and the workspace manager orchestrates interactions between applications.

2.4 Twelve Core Applications

Application	Primary Purpose	File Integration
Notes Editor	Writing, documentation, focus work	Reads/writes text files
Timer/Pomodoro	Time management, focus sessions	None
Todo Manager	Task tracking, project management	Stores task lists
Code Editor	Programming, technical learning	Reads/writes code files
Quiz/Flashcard	Study, learning, active recall	Stores quiz data
Email Composer	Communication, formal writing	Saves drafts
Chat Interface	Quick messaging, collaboration	Message history
Calendar	Scheduling, time blocking	Event data
File Browser	File navigation, organization	Full file system

Whiteboard	Brainstorming, visual thinking	Saves diagrams
AI Chat Panel	Assistance, learning support	Conversation history
Explanation Panel	Contextual help, education	None

3. State Management Strategy

3.1 Separation of Concerns

The architecture enforces strict separation between four distinct state domains to prevent coupling and maintain clarity. Each domain has specific responsibilities and should never be mixed with others.

Domain	Scope	Storage Method	Responsibilities
Workspace State	Global	Zustand Store	Current configuration, active modes, app positions, workspace history
Application State	Per-App	Component State	Internal app data, timer countdown, note content, task lists
File System State	Global	Separate Store	File tree, open files, recent files, metadata cache
UI State	Local	Component State	Drag operations, resize handles, dropdowns, hover states

3.2 Unidirectional Data Flow

The application implements a strict unidirectional data flow pattern where the workspace state serves as the single source of truth. Applications react to state changes rather than mutating state directly. User interactions flow upward through event handlers, trigger state updates, and cause re-renders that reflect the new state.

3.3 State Persistence Strategy

State Domain	Persistence Method	Persistence Trigger
Workspace Configuration	Browser localStorage	On every workspace change
Workspace History	Browser localStorage	When new workspace created
File Contents	IndexedDB	On every file save operation
File Metadata	IndexedDB	When file created or modified
User Preferences	Browser localStorage	On preference change
UI State	None (transient)	Not persisted

4. File Management Architecture

4.1 Virtual File System Abstraction

The system implements a complete virtual file system abstraction that mimics a traditional operating system file system while storing all data in the browser. This abstraction provides applications with a familiar file system interface while maintaining flexibility in the underlying storage implementation.

4.2 File System Design Principles

Files are identified by absolute paths following Unix-style conventions, such as /projects/code.js or /notes/meeting.txt. Directories exist conceptually to organize files but may not be physically stored separately. Applications interact exclusively through the file system API and remain unaware of the underlying storage mechanism, enabling the storage implementation to be changed without affecting applications.

4.3 Three-Layer File System Architecture

Layer 1: File System API (Public Interface)

Operation	Purpose	Return Value
readFile(path)	Retrieve file contents	File content as string or binary
writeFile(path, content)	Save or update file	Success confirmation
listDirectory(path)	Get directory contents	Array of file/directory names
deleteFile(path)	Remove file	Success confirmation
moveFile(oldPath, newPath)	Relocate file	Success confirmation
getMetadata(path)	Retrieve file info	Size, modified date, type

Layer 2: Storage Adapter (Implementation)

The storage adapter translates API calls into IndexedDB operations, handles file metadata management including modification dates and file sizes, maintains the directory tree structure, and can be replaced with alternative implementations such as cloud storage without affecting the applications that depend on it.

Layer 3: IndexedDB (Browser Primitive)

The lowest layer utilizes browser IndexedDB with two object stores: one for files and one for directories. Files are keyed by their complete path, enabling efficient retrieval. Directory listings are implemented through path prefix queries that match all files beginning with a specific directory path.

4.4 File Reference Pattern in Applications

Applications store file paths rather than file contents in their internal state. For example, the Notes application stores the path /notes/current.txt rather than the note content itself. When the component renders, it fetches the content from the file system. When changes occur, the application writes the updated content back to the file system through the API.

5. Drag and Drop System

5.1 Window Dragging Implementation

The window dragging system enables users to reposition application windows through mouse interactions. The system operates through three distinct event phases that coordinate to provide smooth, responsive dragging behavior.

Phase	Trigger	Actions
Initiation	Mouse down on title bar	Capture initial cursor position, record starting window coordinates, enter dra
Movement	Mouse move while dragging	Calculate position delta, update application position in workspace state, con
Completion	Mouse up anywhere	Exit drag mode, save final position to workspace state, persist to localStorag

5.2 Constraint System

The dragging system implements several constraints to maintain usability and visual consistency. Applications cannot be dragged beyond the viewport boundaries, preventing windows from becoming inaccessible. An optional snap-to-grid feature rounds positions to the nearest ten-pixel increment for alignment. Magnetic edge detection activates when an application edge comes within five pixels of another application, causing automatic alignment.

5.3 Window Resizing Mechanism

Each application window provides eight resize handles located at the four corners and four edge midpoints. Different handles modify different dimensions: the bottom-right handle increases both width and height, the top-left handle adjusts both position and dimensions simultaneously, and edge handles modify only width or height. The system enforces minimum dimensions of 200 by 150 pixels to ensure applications remain usable.

5.4 Performance Optimization

During drag operations, the system performs optimistic state updates to ensure smooth visual feedback without waiting for state management operations to complete. Position updates occur on every mouse move event for immediate visual response. Only when the drag completes does the system commit the final position to the persistent workspace state and localStorage.

6. System Modes Architecture

6.1 Modes as Global Modifiers

System modes represent cross-cutting concerns that affect all applications simultaneously rather than functioning as standalone applications themselves. Modes modify the behavior and appearance of the entire workspace, creating different working environments suitable for various contexts.

6.2 Three Essential System Modes

Mode	Visual Changes	Behavioral Changes	Use Cases
Focus Mode	Dim non-essential UI, minimal chrome, subdued colors	Disable notifications, block interruptions, optionally hide inactive windows	Deep work, studying, writing, coding sessions
Dark Mode	Dark background colors, light text, reduced blue light	None (purely visual)	Night work, eye strain reduction, user preference
Do Not Disturb	Status indicator visible, modal blocker overlay	Block external notifications, optionally queue messages	Meetings, presentations, timed focus

6.3 Mode Implementation Strategies

Strategy 1: CSS Class Application

For visual modes such as Dark Mode, the system adds CSS classes to the root HTML element. Applications respond to these classes through CSS rules that adjust colors, fonts, and visual styling. This approach requires no JavaScript in applications and provides consistent theming across all components.

Strategy 2: React Context Provider

For behavioral modes such as Focus Mode, the system uses a React Context that wraps the entire workspace. Applications consume this context to determine active modes and adjust their behavior accordingly. This enables applications to hide notifications, reduce animations, or modify functionality based on the current mode.

Strategy 3: Overlay Components

Some modes render additional overlay components that appear above all applications. Focus Mode may render a dimming overlay that reduces the visibility of inactive applications. Do Not Disturb displays a persistent status indicator. These overlays exist independently of applications and are managed by the workspace container.

6.4 Mode Composition and Stacking

Multiple modes can be active simultaneously, creating compound effects. The combination of Dark Mode and Focus Mode provides both visual theming and behavioral changes. Applications check for mode presence using array inclusion tests and apply all relevant

modifications. The workspace state maintains modes as an array of active mode identifiers.

7. Layout Strategies

7.1 Layout Algorithm Overview

The layout engine supports multiple arrangement algorithms that determine how applications are positioned within the workspace. The choice of layout strategy affects the visual organization and spatial relationships between applications, enabling different working styles.

7.2 Four Core Layout Strategies

Strategy	Characteristics	Best For	Implementation
Floating	Manual positioning, free placement, overlapping allowed	Custom arrangements, complex workflows	Direct coordinate specification from AI or user
Grid	Snap to grid cells, defined columns and rows	Organized layouts, quick setup	Divide viewport into cells, assign apps to cells
Split	Binary divisions, proportional sizing, regions	Side-by-side comparisons, primary + secondary layouts	Recursive binary subdivision
Tiled	Automatic arrangement, binary space partitioning, no overlap	Maximum screen usage, balanced layouts	Recursive binary split based on app count

7.3 Layout Engine Responsibilities

When a workspace is created, the layout engine receives the list of applications from the intent parser, applies the selected layout strategy to calculate coordinates, and returns positioned applications to the workspace state. When the browser window is resized, the engine recalculates positions proportionally to maintain relative layouts while adapting to the new dimensions.

7.4 Dynamic Layout Switching

Users can switch between layout strategies without recreating the workspace. The layout engine recalculates positions based on the new strategy while preserving the set of active applications. This enables users to experiment with different arrangements to find the most effective organization for their current task.

8. Data Flow Architecture

8.1 Complete System Data Flow

Step	Component	Action	Data Passed
1	User	Types natural language intent	Intent string
2	Intent Input Component	Sends API request	JSON with intent
3	Next.js API Route	Calls Gemini API	Intent + system prompt
4	Gemini AI	Parses and structures intent	Workspace configuration JSON
5	API Route	Returns to client	Structured workspace config
6	Workspace Store	Updates state	Apps array, modes array, layout type
7	Layout Engine	Calculates positions	Positioned app configurations
8	App Renderer	Instantiates components	App props with coordinates
9	Mode System	Applies global modifiers	Active modes array
10	Applications	Render with modes	UI components
11	File System	Loads app data	File contents from IndexedDB
12	User	Interacts with apps	Edits, clicks, drags
13	Applications	Update local state	Modified data
14	File System	Persists changes	Updated files to IndexedDB
15	Workspace Store	Saves configuration	Updated positions to localStorage

8.2 State Update Patterns

The system maintains unidirectional data flow where workspace state serves as the single source of truth. Applications react to state changes through props updates. User interactions trigger events that flow upward to state management. State updates cause re-renders that propagate changes downward to all affected components.

9. Implementation Roadmap

9.1 Phase 1: Foundation (Weeks 1-2)

Core Infrastructure Development

Component	Tasks	Success Criteria
Next.js Setup	Initialize project, configure TypeScript, setup folder structure	Project builds successfully
Server Components	Create API route for intent parsing, integrate Gemini API, implement error handling	API returns valid JSON
State Management	Setup Zustand store, implement workspace state, add persistence	State persists across page refreshes
File System	Create IndexedDB wrapper, implement basic file operations, build file system API	Files can be created, read, written
Core Apps (5)	Implement Notes, Timer, Todo, AI Chat, Email Composer	Apps render and function independently

9.2 Phase 2: Expansion (Weeks 3-4)

Feature Completion and Integration

Component	Tasks	Success Criteria
Remaining Apps (7)	Code Viewer, Quiz, Calendar, File Browser, Whiteboard, Camera, Explanation Panel	All 12 apps functional
Layout Engine	Implement floating layout, add grid layout, create split layout, layout switching	Multiple layout strategies working
Drag and Drop	Window dragging, resize handles, constraint system, snap-to-grid	Smooth window manipulation
System Modes	Focus Mode implementation, Dark Mode styling, Mode-aware apps	Modes affect all apps correctly
App Registry	Central app mapping, metadata system, dynamic imports	Apps load dynamically by ID

9.3 Phase 3: Polish and Intelligence (Weeks 5-6)

User Experience Enhancement

Component	Tasks	Success Criteria
AI Enhancement	Improve intent parsing, add mode suggestions, implement workspace recommendations	90% intent accuracy
Workspace Management	Workspace history, quick recall, favorites system	Users can save and reload workspaces
Performance	Optimize rendering, lazy load apps, reduce re-renders	Under 2 second workspace creation
Testing	User testing sessions, bug fixes, UI refinement	Positive user feedback
Documentation	API documentation, developer guide, user manual	Complete documentation

10. Critical Architectural Decisions

10.1 Technology Stack Rationale

Technology	Rationale	Alternatives Considered
Next.js	Server components for API security, built-in API routes, excellent TypeScript support, fast refresh during development	Vite + React (lighter but more setup), Remix (newer, smaller ecosystem)
Zustand	Minimal boilerplate, excellent TypeScript support, simple API, persistence	Redux (too verbose), Context API (performance issues)
IndexedDB	Large storage capacity (50GB+), works offline, no size limits, client-side	localStorage (size limits), cloud storage (requires backend)
TypeScript	Type safety for complex state, better developer experience, catches errors early	JavaScript (more bugs), JSDoc (less powerful, harder refactoring)

10.2 Design Patterns and Principles

Principle	Application	Benefit
Separation of Concerns	Four distinct state domains, each with clear responsibilities	Easier debugging, independent testing, clear boundaries
Unidirectional Data Flow	State flows down through props, events flow up through callbacks	Predictable behavior, easier reasoning, simpler debugging
Composition over Inheritance	Modes and apps combine through composition, not inheritance	Flexible combinations, flexible system, maintainable code
Single Source of Truth	Workspace state is canonical, apps derive from it	Consistency, no sync issues, clear data ownership
Interface Segregation	Apps use minimal file system API, not implementation details	Swappable implementations, cleaner code, better abstraction

10.3 Scalability Considerations

The architecture is designed to scale in multiple dimensions. Additional applications can be added to the registry without modifying existing code. New system modes can be implemented by adding CSS classes and context values. Storage can be migrated from IndexedDB to cloud solutions by swapping the storage adapter while maintaining the same API. Layout strategies can be extended by implementing new layout algorithms.

11. Conclusion

11.1 Summary of Key Achievements

This architecture specification defines a comprehensive, production-ready system for building the Intent-Driven OS on Next.js. The design achieves several critical objectives: it maintains clear separation between server and client components, implements a scalable state management strategy, provides a complete browser-based file system, enables smooth drag-and-drop interactions, and supports flexible system-wide modes.

11.2 Architectural Strengths

Strength	Description
Modularity	Apps, modes, and layout strategies are independently developed and composed
Flexibility	12 apps and 3 modes create 70+ unique workspace combinations
Performance	Browser-based storage eliminates network latency for file operations
Security	API keys stay on server, Gemini calls never expose credentials
User Experience	Natural language intent creates workspaces in under 2 seconds
Maintainability	Clear separation of concerns enables isolated development and testing
Scalability	New apps and modes can be added without modifying core architecture

11.3 Future Enhancement Opportunities

While the MVP architecture is complete and production-ready, several enhancement opportunities exist for future development. Cloud synchronization can be added by implementing a storage adapter that syncs IndexedDB with cloud storage. Collaborative features can be built on top of the existing workspace system. Additional layout strategies such as automatic tiling or circular arrangements can be implemented. More sophisticated AI features including workspace learning and predictive app suggestions can enhance the intelligence layer.

11.4 Next Steps

Development should proceed according to the three-phase implementation roadmap outlined in Section 9. Phase 1 establishes the foundational infrastructure and implements five core applications. Phase 2 completes the application suite and implements all interaction systems. Phase 3 adds intelligence features and performs user testing and refinement. Following this roadmap ensures a systematic approach that delivers value incrementally while maintaining architectural integrity.

11.5 Final Remarks

This architecture represents a thoughtful balance between ambition and pragmatism. It provides the flexibility to create diverse, intelligent workspaces while maintaining implementation simplicity. The system is designed to feel adaptive and intelligent without becoming complex or overwhelming. By organizing around human intent rather than traditional application boundaries, the Intent-Driven OS creates a fundamentally different interaction paradigm that prioritizes user goals over technical structures.