# Architectural Blueprint for the Modular "Command Center" ERP

## Executive Summary

The enterprise software landscape is undergoing a tectonic shift from rigid, monolithic forms to malleable, composable canvases. The modern organization no longer accepts the dichotomy between "Documents" (unstructured text in Word/Google Docs) and "Applications" (structured data in Salesforce/SAP). The "Command Center" ERP vision represents the convergence of these two worlds: a "Smart Doc" interface where the narrative flexibility of a document serves as the container for the rigorous transactional logic of an ERP. This report details the architectural foundation for such a system, designated hereafter as the **Modular Command Center (MCC)**.

This analysis proceeds from the role of a Principal Software Architect tasked with delivering a robust, scalable Version 1.0 "Skeleton." The primary directive is "Extreme Modularity." The system is not a single application but a host environment—a "Browser for Business Logic"—capable of loading disparate business units (CRM, Finance, Inventory) as pluggable modules.

Our architectural strategy leverages **Next.js 14** for the application framework, **Supabase** (PostgreSQL) for the persistence and security layer, and **BlockNote** (built on TipTap/ProseMirror) for the rich text interface. This triad offers the optimal balance of type safety, developer velocity, and deep extensibility. By adopting a "Modular Monolith" architecture, we avoid the premature complexity of microservices while strictly enforcing domain boundaries through the Next.js App Router and dynamic component registries.

The following sections provide an exhaustive technical analysis of every subsystem, from the jsonb indexing strategies in PostgreSQL required to query "live" widgets, to the dynamic import patterns in React necessary to keep bundle sizes manageable. The report concludes with three precise implementation documents: the Project Definition, the Version 1.0 Specification, and the .cursorrules for AI-assisted development.

## 1. The Paradigm Shift: From Monoliths to Composable Canvases

### 1.1 The Evolution of the ERP Interface

Historically, Enterprise Resource Planning (ERP) systems were designed around the limitations of relational databases. The user interface was a direct projection of the database schema: a "Customers" table became a "Customer List" page, and a "Customer" row became a "Customer Detail" form. This "Database-First" design philosophy resulted in rigid silos where data was trapped in specific views. To connect a "Project Plan" (unstructured text) with a "Budget" (structured data), users had to resort to screenshots or hyperlinks between disparate systems.

The "Command Center" vision rejects this obsolescence. It embraces the **"Everything is a Block"** philosophy popularized by Notion and Coda. In this paradigm, the fundamental unit of the application is not the "Page" or the "Form," but the **Block**. A paragraph of text is a block. A checklist is a block. Crucially, a live, interactive Kanban board displaying real-time inventory data is also just a block.

This shift necessitates a "Document-First" architecture. The frontend is no longer a static set of routes; it is a dynamic renderer capable of interpreting a serialized document structure (JSON) and "hydrating" it with interactive React components on the fly.1 The implications for the backend are equally profound: the database must support highly granular, mixed-content storage where relational integrity coexists with document-oriented flexibility.

### 1.2 The "Coda-Like" Experience Requirements

To successfully replicate the "Coda-like" experience, the MCC must satisfy three non-negotiable UX requirements, which drive specific architectural decisions:

1. **Fluidity of Narrative and Logic:** Users must be able to write a strategy document (Narrative) and embed a live revenue chart (Logic) in the same flow. This requires a Rich Text Editor (RTE) that supports "Custom Node Views".3
2. **Live Interactivity:** The embedded widgets are not static screenshots. They are fully functional mini-applications. A user must be able to update a CRM lead's status directly from the document. This demands a robust client-side state management strategy, leveraging tools like **TanStack Query** to bridge the gap between the document's ephemeral state and the database's persistent state.4
3. **Infinite Extensibility:** The system cannot ship with every possible widget pre-installed. It needs a plugin architecture where new modules (e.g., a "GitHub Issues" widget) can be added without rewriting the core editor. This points directly to the **Registry Pattern** and **Dynamic Imports** in Next.js.5

### 1.3 Architectural Priorities for Version 1.0

The "Strong Skeleton" priority mandates that V1.0 focuses on the *capabilities* of the system rather than the *breadth* of features. We are building the engine, not the car. The critical path involves:

* **The Render Loop:** Can the system reliably save a complex document structure to Supabase and re-render it exactly as it was?
* **The Security Model:** Can we enforce Row Level Security (RLS) such that a user can see the document layout but be denied access to sensitive data within a specific widget?
* **The Plugin Interface:** Is the developer experience (DX) for creating a new widget standardized and type-safe?

## 2. The Core Engine: BlockNote & The "Smart Doc" Architecture

### 2.1 The Editor Engine Decision: BlockNote vs. TipTap

The choice of the underlying editor engine is the single most critical technical decision for the MCC. The research landscape presents two primary contenders: **BlockNote** and **TipTap**.

| **Feature** | **BlockNote** | **TipTap** | **Architectural Implication** |
| --- | --- | --- | --- |
| **Foundation** | Built on TipTap & ProseMirror | Built on ProseMirror | Both share the same robust underlying model. |
| **Abstraction Level** | High (Batteries Included) | Low (Headless Framework) | BlockNote offers speed; TipTap offers control. |
| **Block Logic** | Native (Drag & Drop, Nesting) | Manual Implementation | TipTap requires weeks of effort to rebuild "Notion-like" block interactions. |
| **Customizability** | Structured Extensions | Total DOM Control | BlockNote is strictly React-focused.7 |
| **V1 Suitability** | **High** | Medium | BlockNote validates the "Coda" feel immediately. |

**Strategic Decision:** For V1.0, **BlockNote** is the superior choice. The "Strong Skeleton" mandate requires us to validate the complex interactions of a block-based editor (nesting, drag-and-drop reordering, slash menus) without dedicating months of engineering time to reinventing the wheel. BlockNote provides this "Notion-like" behavior out-of-the-box.1

Crucially, BlockNote is not a "black box" that prevents deeper access. It exposes the underlying TipTap/ProseMirror instance, allowing us to drop down to the lower level if necessary. However, relying on BlockNote means we accept an abstraction layer. To mitigate the risk of "outgrowing" BlockNote, we will encapsulate the editor component within a CommandCenterEditor wrapper, ensuring that the rest of the application interacts with our interface, not the library directly.

### 2.2 The Custom Block Architecture

The core differentiator of the MCC is the ability to insert "Smart Blocks" (Widgets). In BlockNote, this is achieved through **Custom Schemas** and **React Node Views**.9

#### 2.2.1 The Schema Definition

Every widget acts as a custom block type in the ProseMirror schema. This requires defining:

1. **The Name:** A unique string ID (e.g., cc-widget-revenue).
2. **The Attributes:** The configuration state of the widget (e.g., { "chartType": "bar", "dataSource": "q3\_revenue" }).
3. **The Rendering Logic:** Mapping the block to a specific React Component.

The schema definition must be strict. We utilize TypeScript to enforce that the attributes object matches the configuration props expected by the React component. This prevents runtime errors where a widget attempts to render with missing configuration data.

#### 2.2.2 State Management: The "Two-Layer" State

A "Smart Doc" introduces a complex state management challenge. The state exists in two distinct layers that must be synchronized but decoupled:

1. **Document State (Layout & Config):** This is the JSON structure managed by BlockNote. It stores *what* widgets are present and *how* they are configured. It is persistent and versioned.
   * *Example:* Block ID: 123, Type: RevenueWidget, Config: { year: 2024 }
2. **Widget State (Transactional Data):** This is the live data displayed *inside* the widget. It is ephemeral and fetched on-demand.
   * *Example:* Revenue: $1,500,000

**Architectural Rule:** The Document State **never** stores transactional data. We do not save the "$1,500,000" figure into the document JSON. Instead, the widget uses the Config to query the Supabase backend via React Query at runtime. This ensures that the document always reflects the "Single Source of Truth" in the database and prevents stale data from being frozen in time within a document blob.

### 2.3 React Node Views & Performance

Rendering heavy React components (like data grids or charts) inside a text editor can lead to performance bottlenecks. ProseMirror re-renders node views frequently.

**Optimization Strategy:**

1. **ReactNodeViewRenderer:** We utilize TipTap's React renderer which optimizes the binding between the DOM and React's Virtual DOM.10
2. **contentDOM Separation:** For widgets that contain editable text *inside* them, we must carefully manage the contentDOM prop to ensure ProseMirror can manage the cursor position without React interfering.10
3. **Memoization:** All widget components must be wrapped in React.memo. They should only re-render if their props.node.attrs (Configuration) change, not when the user types in a paragraph three blocks down.

## 3. Frontend Architecture: Next.js 14 & The Modular Monolith

### 3.1 The "Feature-Sliced" Modular Monolith

To support "Infinite Integrations," the Next.js application must be structured to prevent the "Spaghetti Monolith" antipattern. We will adopt a **Feature-Sliced Design** approach, organizing the codebase by *Business Domain* rather than *Technical Layer*.12

#### 3.1.1 Directory Structure

The standard Next.js app/ directory is reserved strictly for **Routing** and **Layouts**. All business logic resides in a parallel modules/ directory.

Plaintext

/src  
 /app # ROUTING LAYER  
 /(dashboard)  
 /documents/[id]/page.tsx  
 /modules # DOMAIN LAYER  
 /editor # The Core Editor Module  
 /components  
 EditorWrapper.tsx # The BlockNote implementation  
 Registry.ts # The Dynamic Widget Registry  
 /crm # A Business Unit  
 /components  
 LeadListWidget.tsx  
 /actions.ts # Server Actions for CRM data  
 /types.ts  
 /finance # Another Business Unit  
 /components  
 RevenueChartWidget.tsx

This structure enforces modularity. The crm module contains everything it needs: its UI components, its backend Server Actions, and its type definitions. If we need to remove the CRM feature, we simply delete the src/modules/crm folder and remove one line from the Registry.

### 3.2 The Dynamic Component Registry

The mechanism that links the "Document JSON" to the "Modular Code" is the **Component Registry**. When the editor encounters a block type crm-lead-list, it needs to load the corresponding code.

Statically importing all widgets into the Editor would be catastrophic for performance. It would create a massive initial JavaScript bundle containing the code for every possible widget in the system.

Solution: The Lazy Registry Pattern

We utilize Next.js's dynamic import function to implement code splitting at the widget level.5

TypeScript

// src/modules/editor/registry.ts  
import dynamic from 'next/dynamic';  
import { SkeletonWidget } from '@/components/ui/skeleton';  
  
export const WIDGET\_REGISTRY = {  
 'crm-lead-list': dynamic(() => import('@/modules/crm/components/LeadListWidget'), {  
 loading: () => <SkeletonWidget title="Loading Leads..." />,  
 ssr: false // Widgets interact with browser APIs  
 }),  
 'finance-revenue': dynamic(() => import('@/modules/finance/components/RevenueChartWidget'), {  
 loading: () => <SkeletonWidget title="Loading Chart..." />,  
 ssr: false  
 }),  
};

This pattern ensures that the code for RevenueChartWidget is *only* fetched from the server if the user opens a document that actually contains that widget. This is essential for the "Infinite Integrations" requirement; we can have 1,000 available widgets without impacting the initial load time of the application.14

### 3.3 Server Components vs. Client Components

Next.js 14's React Server Components (RSC) paradigm requires a distinct separation of concerns.

* **The Editor (Client Component):** The BlockNote editor relies heavily on document, window, and Selection APIs. It acts as a Client-Side Single Page Application (SPA) embedded within the page. It must be marked with 'use client'.15
* **The Page Shell (Server Component):** The wrapping page (page.tsx) is a Server Component. It is responsible for the initial data fetch: retrieving the Document JSON and the User's RBAC permissions from Supabase. It passes this data as props to the Client Editor.
* **The Widgets (Client Components):** While it is theoretically possible to stream Server Components into a Client Component, the interactive nature of the widgets (sorting, filtering, live updates) makes them better suited as Client Components fetching data via **TanStack Query**.4 This allows for robust client-side caching, polling for real-time updates, and optimistic UI mutations—critical for a "Command Center" feel.

## 4. The Data Backbone: Supabase & PostgreSQL

### 4.1 Schema Design: The Hybrid Model

Designing the database schema for a "Notion-clone" involves a fundamental trade-off between **JSONB** (Document) and **Relational** (SQL) models.

* **Option A: The Blob (JSONB).** Store the entire document content as a single JSON object.
  + *Pros:* Fast reads/writes. Trivial version history. Matches the frontend model exactly.
  + *Cons:* Difficult to query "inside" the document (e.g., "Find all docs with a Revenue Widget").
* **Option B: Atomic Blocks (Relational).** Store every paragraph as a row in a blocks table.
  + *Pros:* Granular queryability. Granular permissions.
  + *Cons:* Heavy write load (saving a doc updates 50 rows). Complex sequencing logic.

Recommendation: The Hybrid "Metadata Extraction" Model

For V1.0, we prioritize performance and developer velocity. We will store the content as a JSONB blob but extract critical metadata into relational tables to support the "Command Center" query requirements.16

**Table Structure:**

| **Table Name** | **Description** | **Key Columns** |
| --- | --- | --- |
| documents | Stores the document entity and the full content blob. | id (UUID), content (JSONB), org\_id (UUID) |
| document\_widgets | An index table tracking which widgets are used in which docs. | doc\_id (FK), widget\_type (Text), widget\_config (JSONB) |
| workspaces | The tenant isolation unit. | id (UUID), name (Text), settings (JSONB) |
| members | Users belonging to workspaces. | user\_id (FK), workspace\_id (FK), role (Enum) |

The Write Mechanism:

When the Editor saves a document (via a Supabase Edge Function or Next.js Server Action), the backend logic parses the content JSON. It identifies all "Block" objects with a type of widget-\*. It then updates the document\_widgets table.

* *Query Benefit:* To find "All documents containing the Q3 Revenue Chart," we query the small, indexed document\_widgets table, rather than scanning the massive documents table.18

### 4.2 Optimizing JSONB with Indexing

Even with the hybrid model, we may need to query the JSONB blob directly. PostgreSQL offers robust indexing for this via **GIN (Generalized Inverted Index)**.20

We will apply a jsonb\_path\_ops GIN index to the content column:

SQL

CREATE INDEX idx\_documents\_content ON documents USING GIN (content jsonb\_path\_ops);

This index is highly efficient for containment queries (@>). While BlockNote's JSON structure is deeply nested (blocks inside children arrays), PostgreSQL's JSONB operators allow us to traverse this structure efficiently if needed.21

### 4.3 Row Level Security (RLS) & Multi-Tenancy

Security in a B2B SaaS must be enforced at the database level. Supabase RLS is the ideal mechanism. We will enforce strict multi-tenancy based on the workspace\_id.

The Security Context:

Every query to the database usually carries the authenticated user's JWT. We extract the user's ID (auth.uid()) and map it to their workspace memberships.

**RLS Policy Implementation:**

SQL

-- Policy: Access to Documents  
CREATE POLICY "Users can access documents in their workspace"  
ON documents  
FOR ALL  
USING (  
 workspace\_id IN (  
 SELECT workspace\_id   
 FROM members   
 WHERE user\_id = auth.uid()  
 )  
);

This single policy guarantees that a user can never access data from another organization, regardless of bugs in the frontend application code.22

## 5. Security & Access Control: The Granular Defense

### 5.1 The "Document vs. Data" Security Gap

A unique challenge of the "Command Center" architecture is that a user might have permission to view a *Document* but lack permission to view the *Data* inside a specific widget on that document.

* *Scenario:* A generic "Company Update" document contains a "Sensitive HR Issues" widget.
* *Risk:* If the HR data is baked into the document JSON, any user who can read the document can read the HR data.

The Solution: Decoupled Authorization

This reinforces the "Two-Layer State" architectural rule.

1. **Document Access:** Controlled by the documents table RLS. If the user accesses the document, they download the JSON layout. They see that a widget *exists* at a specific location.
2. **Widget Data Access:** Controlled by the specific table RLS queried by the widget (e.g., hr\_tickets table).
   * When the "HR Widget" component mounts, it attempts to fetch data from hr\_tickets.
   * If the user is not an HR admin, the RLS policy on hr\_tickets returns an empty set or an error.
   * The widget handles this error gracefully, rendering a "Access Denied" state (e.g., a blurred placeholder).
   * **Result:** The layout is preserved, but the sensitive data never leaves the database.24

### 5.2 Secure Component Loading

Allowing "pluggable modules" introduces a theoretical risk of malicious code injection if we were to allow third-party developers to upload raw JavaScript. For V1.0, the "Modules" are internal, built by our team. However, the architecture anticipates this future risk.

* **V1 Mitigation:** All modules are part of the compiled Next.js bundle. No dynamic code execution (eval) is permitted.
* **V2 Roadmap:** For external plugins, we will need a sandboxed execution environment (like QuickJS in WebAssembly) or an iframe-based isolation model similar to Figma plugins.

## 6. Implementation Strategy: The V1 Skeleton

### 6.1 Phase 1: The Foundation (Weeks 1-3)

The priority is establishing the "Strong Skeleton." This involves setting up the Next.js repo, the Supabase project, and the basic BlockNote editor.

* **Key Deliverable:** A user can log in, create a workspace, create a document, type standard text, and save it. The data persists to Supabase.

### 6.2 Phase 2: The Registry & Custom Blocks (Weeks 4-5)

We implement the WidgetRegistry and the dynamic import logic. We create a "Reference Implementation" widget—a simple "Counter" or "Todo List" that stores its state in a separate Supabase table, proving the decoupled data model.

* **Key Deliverable:** The /widget slash command works. A custom React component renders inside the editor.

### 6.3 Phase 3: The Dashboard Fusion (Weeks 6-8)

We build the first "Real" business module (e.g., a CRM module). We implement the "Index Table" logic (document\_widgets) to track widget usage. We refine the RLS policies to ensure the Document/Data security separation works as intended.

* **Key Deliverable:** A functioning "Command Center" where a user can build a dashboard using text and live CRM widgets.

# Appendix A: Project Definition (project\_definition.md)

# Project Definition: Modular Command Center ERP

## 1. Vision Statement

To architect and build a scalable "Command Center" ERP that fuses the narrative flexibility of a document editor with the transactional power of a business dashboard. The system empowers users to compose their own workflows by embedding "live" business units (Widgets) directly into smart documents, replacing rigid, siloed ERP interfaces with a malleable, composable canvas.

## 2. Core Architectural Pillars

### 2.1 Extreme Modularity (The "Browser" Concept)

The core application acts as a host environment—a browser for business logic. Features such as CRM, Finance, and HR are built as isolated **Modules**. The core system provides the services (Auth, Navigation, Editor Shell), while the Modules provide the content (Widgets, Data Actions). This is implemented via a **Feature-Sliced Modular Monolith** architecture.

### 2.2 The "Strong Skeleton" (V1.0 Priority)

Version 1.0 is not a feature-complete ERP. It is the robust architectural skeleton required to support infinite future integrations. The priority is validating the end-to-end flow of:

1. **Composition:** Inserting a dynamic React component into a serialized JSON document.
2. **Persistence:** Saving mixed-content (text + widget config) to PostgreSQL.
3. **Hydration:** Loading and rendering the document with live data fetching.
4. **Security:** Enforcing granular RLS across both document layouts and widget data.

## 3. Technology Stack

* **Frontend Framework:** Next.js 14 (App Router)
* **Language:** TypeScript (Strict Mode)
* **Data & Auth:** Supabase (PostgreSQL, GoTrue)
* **Editor Engine:** BlockNote (wrapping TipTap/ProseMirror)
* **State Management:** TanStack Query (React Query)
* **Styling:** Tailwind CSS + shadcn/ui (Radix Primitives)
* **Package Manager:** pnpm

## 4. User Personas

* **The Architect (Admin/Manager):** Responsible for designing workflows. They create the "Master Docs," configuring widgets and permissions. They require deep control over data sources and layout.
* **The Operator (Contributor):** The daily user. They open documents to update specific data points (e.g., "Move Lead to Qualified"). They require speed, clarity, and zero friction.

## 5. Risk Assessment & Constraints

* **Performance Risk:** A document with 50 live widgets could cause network congestion.
  + *Mitigation:* Implementation of "Lazy Hydration" via Intersection Observer and React Query. Widgets only fetch data when scrolled into view.
* **Security Risk:** Data leakage via document sharing.
  + *Mitigation:* Strict decoupling of Document RLS (Layout) and Widget RLS (Data).
* **Complexity Risk:** The "Block" logic of rich text editors is notoriously difficult.
  + *Mitigation:* Adoption of BlockNote to abstract the low-level ProseMirror complexity for V1.

## 6. Success Criteria (V1.0)

1. **Auth:** Secure Multi-tenant Login (Email/Password + OAuth).
2. **Editor:** Functional Rich Text Editor with slash commands.
3. **Registry:** Capability to insert at least one "Custom Widget" via the registry.
4. **Persistence:** Documents persist to DB and reload correctly.
5. **Security:** User A cannot access User B's documents (verified via RLS tests).

# Appendix B: Version 1.0 Specification (version\_1.0.md)

# Version 1.0 Implementation Specification: The Strong Skeleton

## 1. System Architecture

### 1.1 Directory Structure (Modular Monolith)

The project adheres to a strict domain-driven structure to facilitate future extraction of modules.bash

/src

├── /app # NEXT.JS ROUTING LAYER (Thin Wrappers)

│ ├── /(auth) # Public authentication routes

│ ├── /(dashboard) # Protected application shell

│ │ ├── /documents # Document viewer/editor pages

│ │ └── /settings # Workspace configuration

│ └── /api # Route Handlers (Edge Functions)

├── /components # SHARED UI PRIMITIVES (shadcn/ui)

│ ├── /ui # Atomic components (Button, Input, Dialog)

│ └── /layout # Shell components (Sidebar, Header)

├── /lib # CORE UTILITIES

│ ├── /supabase # Typed Supabase Client

│ ├── /utils # Helper functions (cn, formatters)

│ └── /hooks # Global hooks (useAuth, useDebounce)

├── /modules # BUSINESS DOMAIN MODULES

│ ├── /core # Foundation (Auth, User Profile, Workspace)

│ ├── /editor # The BlockNote Integration

│ │ ├── /components # EditorWrapper, CustomBlockView

│ │ ├── /schema # Custom Block Definitions

│ │ └── registry.ts # Dynamic Component Registry

│ └── /crm-demo # Proof-of-Concept Module

│ ├── /components # LeadListWidget.tsx

│ └── /actions.ts # Server Actions for Lead Data

└── /types # GLOBAL TYPE DEFINITIONS

└── database.types.ts # Generated Supabase types

## 2. Feature Specifications  
  
### 2.1 The Document Engine (Module: `/editor`)  
\* \*\*Library:\*\* `@blocknote/react` (Client Component).  
\* \*\*Wrapper:\*\* Create `CommandCenterEditor` component to encapsulate BlockNote config.  
\* \*\*Storage:\*\*  
 \* Save logic: `useDebounce` hook (1000ms) triggers `saveDocument` Server Action.  
 \* Data Format: JSONB.  
\* \*\*Custom Block Implementation:\*\*  
 \* Define `WidgetBlockSchema` in BlockNote.  
 \* Implement `WidgetBlockComponent` using `useBlockNote` hook.  
 \* \*\*Registry:\*\* The `registry.ts` file must export a map of `WidgetType -> dynamic(() => import(...))`.  
  
### 2.2 The Data Layer (Supabase)  
\* \*\*Database Schema:\*\*  
 \* `workspaces` (id, name, owner\_id)  
 \* `profiles` (id, email, full\_name, workspace\_id)  
 \* `documents` (id, title, content (jsonb), workspace\_id)  
 \* `crm\_leads` (id, name, status, value, workspace\_id) -- \*For Demo Widget\*  
\* \*\*RLS Policies:\*\*  
 \* Enable RLS on ALL tables.  
 \* Standard Policy: `workspace\_id = (select workspace\_id from profiles where id = auth.uid())`.  
  
### 2.3 The Demo Widget: "CRM Lead List"  
To validate the architecture, V1 must include one fully functional widget.  
\* \*\*Functionality:\*\* A simple table listing leads from `crm\_leads`.  
\* \*\*Interactivity:\*\* Click a "Status" badge to toggle between "New" and "Contacted."  
\* \*\*Data Fetching:\*\* Use `useQuery` to fetch leads. Use `useMutation` to update status.  
\* \*\*Integration:\*\* The widget must be insertable via typing `/leads` in the editor.  
  
## 3. Technical Implementation Details  
  
### 3.1 Dependencies  
\* `next`: ^14.0.0  
\* `@supabase/supabase-js`: ^2.0.0  
\* `@blocknote/react`: Latest  
\* `@tanstack/react-query`: ^5.0.0  
\* `lucide-react`: For icons  
  
### 3.2 Styling Strategy  
\* \*\*Tailwind CSS\*\* for all styling.  
\* \*\*Dark Mode:\*\* Built-in via `next-themes`.  
\* \*\*Typography:\*\* `prose` (Tailwind Typography) class for the editor content area.  
  
### 3.3 Development Guidelines  
\* \*\*Strict Mode:\*\* Enabled. No `any` types permitted.  
\* \*\*Linting:\*\* ESLint + Prettier.  
\* \*\*Git Flow:\*\* Feature branches merged to `main` via PR.  
  
## 4. V1.0 Test Plan  
1. \*\*Auth Flow:\*\* Create new account, create organization.  
2. \*\*Editor Basics:\*\* Write text, use H1/H2/H3, lists, quote.  
3. \*\*Widget Flow:\*\* Insert "CRM Lead List." See mock data. Update a lead. Refresh page. Verify data persists.  
4. \*\*Security:\*\* Log in as User B (different org). Attempt to access User A's document ID. Expect 404 or 403.

# Appendix C:.cursorrules (.cursorrules)

#.cursorrules - AI Coding Assistant Rules

# Role & Context

You are a Principal Software Architect specializing in Next.js 14, Supabase, and Component-Driven Design. You are building a modular "Command Center" ERP (Business Dashboard + Document Editor).

# Architecture & Patterns

1. **Modular Monolith:** STRICTLY adhere to the src/modules/{domain} structure. Do not place business logic in global components or utils.
   * Example: CRM logic goes in src/modules/crm.
   * Example: Auth logic goes in src/modules/core.
2. **Client/Server Split:**
   * Use Server Components (RSC) by default for data fetching (Page Shells).
   * Use Client Components ('use client') for the Editor and Widgets.
   * NEVER import a Server Component directly into the BlockNote editor.
3. **The Registry Pattern:** All editor widgets must be lazy-loaded. Use next/dynamic in src/modules/editor/registry.ts.

# Tech Stack Guidelines

1. **Supabase:**
   * Use database.types.ts for ALL DB interactions.
   * Always handle error from Supabase clients.
   * RLS is the source of truth for security.
2. **React Query:**
   * Use useQuery for fetching data inside Widgets.
   * Use useMutation for writes.
   * Keys must be array-based: ['crm', 'leads', { filter }].
3. **Styling:**
   * Tailwind CSS only.
   * Use cn() (clsx + tailwind-merge) for conditional classes.
   * Use shadcn/ui components for all standard UI elements.

# Coding Standards

1. **TypeScript:** strict: true. No any. Explicitly type all props and return values.
2. **Error Handling:** Use toast (sonner) for user-facing errors. Log technical errors to console.
3. **Performance:**
   * Optimize images with next/image.
   * Ensure dynamic imports have a <Skeleton /> loading state.

# Specific Instructions for BlockNote

* When defining a custom block, separate the **Schema** (logic) from the **Component** (UI).
* Use useBlockNote context to access the editor instance inside widgets.
* Do NOT store heavy transactional data in the Block Attributes. Store only configuration (IDs, filters).

# Tone

Be concise, technical, and precise. Prefer providing code blocks over theoretical explanations.