# Manager’s Packet: Technical Execution Strategy for the "Command Center" ERP (Version 1.0)

## 1. Executive Summary: The "Browser for Business Logic"

### 1.1 Architectural Vision and Strategic Positioning

The "Command Center" ERP represents a fundamental divergence from traditional enterprise software design. Where legacy ERPs force users into rigid, table-centric workflows, and modern documentation tools (Notion, Obsidian) lack rigorous transactional integrity, the Command Center bridges this chasm. We are not merely building a document editor; we are engineering a **Browser for Business Logic**—a host environment where unstructured narrative and structured transactional data coexist in a symbiotic relationship.

As the Principal Systems Architect overseeing the Version 1.0 launch, I have conducted an exhaustive validation of the proposed "Feature-Sliced Modular Monolith" architecture using Next.js 14, Supabase, and BlockNote. The analysis confirms that while the stack is potent, it introduces specific "performance cliffs" inherent to document-oriented data models. Specifically, the tension between the flexibility of JSONB storage and the strict query requirements of an ERP requires a sophisticated "Hybrid Data Model."

This report outlines the execution strategy to mitigate three primary risks identified during deep research:

1. **The "JSONB Swamp":** The tendency for document stores to become unqueryable "data lakes" as they scale.
2. **Bundle Bloat:** The risk of the text editor importing the entire application payload, destroying First Contentful Paint (FCP) metrics.
3. **The "Trojan Horse" Security Gap:** The danger of leaking sensitive transactional data through document layouts via insufficient Row Level Security (RLS) granularity.

The following execution plan mandates the use of PostgreSQL 12+ **Stored Generated Columns** to solve the query performance paradox, a **Strictly Typed Asynchronous Registry** to solve the bundling limitations, and **Dual-Layer Authorization** to ensure military-grade security at the data access level.

### 1.2 Technical Validation & Risk Mitigation

The "Strong Skeleton" philosophy drives V1.0. We are prioritizing the *capabilities* of the system over the *breadth* of features.

The Hybrid Data Model:

Research into PostgreSQL performance 1 indicates that querying deep inside JSONB columns using standard operators (->>) prevents index usage and necessitates full table scans. Furthermore, when JSONB blobs exceed 2KB, PostgreSQL utilizes TOAST (The Oversized-Attribute Storage Technique), compressing the data off the main table heap. Querying a single field from a TOASTed column requires decompressing the entire blob, a catastrophic operation for an ERP dashboard.

* **Resolution:** We will strictly enforce a pattern where the Document JSON stores *only* layout configuration. Transactional data must reside in normalized relational tables. We will bridge the two using **Stored Generated Columns**, which extract metadata from the JSON blob into indexable SQL arrays *on write*, allowing for millisecond-latency aggregation queries without manual trigger maintenance.

The Component Registry Pattern:

To support "Infinite Extensibility," the editor must load widgets dynamically. A naive implementation where the editor imports a registry file that, in turn, imports all widgets will create a "circular dependency hell" and massive bundle sizes.3

* **Resolution:** We will implement a next/dynamic registry that acts as a firewall between the Editor Shell and the Widget Implementation. Barrel files (index.ts) will be strictly prohibited within the widget modules to prevent the accidental inclusion of server-side dependencies in the client bundle.

Dual-Layer Security:

Standard RLS is insufficient if the server renders sensitive data into the document's initial HTML payload. A user might have permission to view a "Company All-Hands" document but lack permission to view the "Q3 Financials" widget embedded within it.

* **Resolution:** The architecture will enforce a strict separation: The Document RLS controls access to the *container* (the existence of the widget), while the Widget RLS controls access to the *content* (the data inside it). Widgets must fetch their own data client-side, ensuring that a lack of permission results in a graceful "Access Denied" UI state rather than a data leak or a page crash.5

## 2. Technical Deep Dive: The Feature-Sliced Modular Monolith

### 2.1 The "Hybrid" Database Architecture

The database schema is the most critical component of the V1.0 skeleton. We are rejecting the pure Document Store model (MongoDB-style) in favor of a PostgreSQL-centric hybrid approach that leverages the relational strengths of Supabase.

#### 2.1.1 The Generated Column Strategy

To enable high-speed filtering (e.g., "Show me all documents that contain a CRM Lead List"), we cannot rely on parsing JSON at query time. We will utilize PostgreSQL's GENERATED ALWAYS AS... STORED syntax.

| **Column Type** | **Definition** | **Usage** | **Performance Impact** |
| --- | --- | --- | --- |
| **JSONB Content** | content jsonb | Stores BlockNote state (layout, text, widget IDs). | High storage, slow query access due to TOAST. |
| **Generated Index** | widget\_index text | GENERATED ALWAYS AS (jsonb\_path\_query\_array(content, '$.\*\*.type')) STORED | **Zero** read overhead. Updates only on write. |
| **Search Vector** | fts tsvector | GENERATED ALWAYS AS (to\_tsvector('english', content ->> 'text')) STORED | Enables full-text search without external services. |

**Reasoning:** This approach moves the computational cost to the INSERT/UPDATE event. When a user saves a document, Postgres automatically parses the JSON, extracts the widget types into a standard SQL array, and indexes them. Queries become simple array containment checks (@>), utilizing standard GIN indexes for blazing-fast performance.2

#### 2.1.2 JSONB Indexing: Path Ops vs. Default

For queries that *must* introspect the JSONB blob, we will utilize the jsonb\_path\_ops GIN operator class instead of the default jsonb\_ops.

* **Technical Justification:** The jsonb\_ops class indexes every key and value separately, creating bloated indexes. jsonb\_path\_ops hashes the path-value pairs. Since our queries will strictly be containment queries (e.g., "Does this document config contain specific filter X?"), jsonb\_path\_ops offers a 30-50% reduction in index size and faster lookup times.8

### 2.2 Next.js 14 Implementation Strategy

The architecture adopts a modified **Feature-Sliced Design (FSD)** to align with the constraints of the Next.js App Router. Standard FSD places the app folder deep within the src structure, which conflicts with Next.js routing conventions.10

#### 2.2.1 Directory Structure & Boundaries

We will enforce a strict separation between **Routing** and **Domain Logic**.

* **src/app**: Contains *only* page shells, layouts, and route handlers. No business logic is permitted here. These files act as "Controllers" in an MVC sense—they fetch data and pass it to modules.
* **src/modules**: This is the heart of the monolith. Each domain (e.g., crm, finance, editor) is self-contained.
  + **Rule:** A module can only import from the public API of another module. Deep imports (e.g., src/modules/crm/internal/helper.ts) are banned via ESLint boundaries.
* **src/components**: Reserved strictly for shadcn/ui primitives (Buttons, Inputs) that have *zero* business logic awareness.

#### 2.2.2 The "No-Barrel" Policy

Research into Next.js performance highlights "Barrel Files" (index.ts files that re-export everything) as a primary cause of slow development builds and large production bundles. Webpack and Turbopack often struggle to tree-shake complex barrel chains effectively, leading to circular dependency errors.3

* **Directive:** We will delete all index.ts files that serve solely as re-exporters. Imports must be explicit: import { Editor } from '@/modules/editor/components/Editor'. This ensures that importing a single icon from a module does not inadvertently load the entire module's dependency graph.

### 2.3 The Extensibility Engine: Dynamic Registry

The core value proposition relies on the ability to load *any* widget without bloating the *core* editor.

#### 2.3.1 The Typed Asynchronous Registry

We will implement a registry pattern that leverages TypeScript's Record type to enforce strict typing on widget keys while using next/dynamic for implementation loading.

**The Implementation Pattern:**

TypeScript

// src/modules/editor/registry.ts  
import dynamic from 'next/dynamic';  
import { ComponentType } from 'react';  
  
// Strict Type Definition for Widget Keys  
export type WidgetType = 'crm-leads' | 'revenue-chart';  
  
// The Registry Map  
export const WIDGET\_REGISTRY: Record<WidgetType, ComponentType<any>> = {  
 'crm-leads': dynamic(() => import('@/modules/crm/components/LeadListWidget'), {  
 loading: () => <div className="h-32 bg-gray-100 animate-pulse" />,  
 ssr: false // CRITICAL: Widgets depend on browser APIs (ResizeObserver, etc.)  
 }),  
 //... future widgets  
};

**Insight:** Setting ssr: false is not just an optimization; it is a stability requirement. Rich text editors and interactive data grids often rely on window or document objects that do not exist on the server. Attempting to SSR these widgets will cause hydration mismatches and server crashes.12

### 2.4 Security Architecture: The Dual-Layer Defense

We must assume that the Client (Browser) is compromised territory. All security enforcement must happen at the Database (RLS) and API (Server Action) layers.

#### 2.4.1 Layer 1: Document Access (Container)

Access to the documents table is controlled by RLS policies linked to the workspace\_id.

* Policy: auth.uid() IN (SELECT user\_id FROM workspace\_members WHERE workspace\_id = documents.workspace\_id).  
  This ensures a user can load the document structure (the JSON blob).

#### 2.4.2 Layer 2: Data Access (Content)

When a widget loads, it executes a useQuery call to fetch transactional data (e.g., from crm\_leads). This fetch request triggers a separate RLS check on the crm\_leads table.

* **Scenario:** User A is in the Workspace but is restricted from "HR" data.
* **Outcome:** They open a document containing an "HR Salaries" widget. They successfully load the document (Layer 1 passes). The widget initializes and requests data. The Database returns a 403 Forbidden for the hr\_salaries table (Layer 2 fails).
* **UX Handling:** The widget must intercept this 403 error and render a "Restricted Access" placeholder, preserving the document layout without leaking data.5

#### 2.4.3 Server Action Security

Server Actions are public API endpoints. We will wrap every Server Action in a higher-order function or middleware that validates:

1. **Authentication:** Is the user logged in?
2. **Authorization:** Does the user belong to the workspace passed in the arguments?
3. Input Validation: Does the input payload match the Zod schema?  
   Using use server does not inherently secure the function; it merely exposes it.14

## 3. The Strategic Roadmap (Weeks 1-8)

This timeline is aggressive but realistic, focused on the "Strong Skeleton" methodology. We build the hardest parts (Architecture, Security, Persistence) first.

### Phase 1: The Foundation & Infrastructure (Weeks 1-2)

**Goal:** Establish the "Walking Skeleton"—Auth, Shell, and Editor State Persistence.

* **Week 1: Infrastructure & Auth**
  + **Repository Setup:** Initialize Next.js 14 with strict ESLint (no barrel files) and Prettier.
  + **Supabase Init:** Configure local development environment with Supabase CLI.
  + **Authentication:** Implement Supabase Auth (Email/Password) and the auth module.
  + **Layout Shell:** Build the responsive sidebar and header using shadcn/ui components.
  + **Data Model V1:** Deploy workspaces, profiles, and documents tables with basic RLS.
* **Week 2: The Core Editor Engine**
  + **BlockNote Integration:** Implement the Editor component using client-only dynamic imports.
  + **Persistence Loop:** Create the saveDocument Server Action. Implement useDebounce to auto-save document JSONB content.
  + **Validation:** Verify that complex nested block structures save and load correctly from PostgreSQL.

### Phase 2: The Registry & Data Hybridization (Weeks 3-4)

**Goal:** Prove the "Hybrid" model and Dynamic Loading capabilities.

* **Week 3: The Registry & Custom Blocks**
  + **Registry Implementation:** Build registry.ts with next/dynamic.
  + **Custom Block Pipeline:** Create a "Hello World" custom block (e.g., an Alert Box) to validate the Zod schema definition and React rendering pipeline.
  + **Generated Columns:** Implement the SQL migration to add widget\_index (Generated Column) to the documents table.
* **Week 4: The CRM Module (Proof of Concept)**
  + **CRM Data Layer:** Create crm\_leads table with separate RLS policies.
  + **CRM Widget:** Build LeadListWidget using TanStack Query.
  + **Integration:** Embed the CRM widget into the Editor. Verify that the document saves the *configuration* (e.g., status='new') while the widget fetches *live data*.

### Phase 3: Advanced Security & Performance (Weeks 5-6)

**Goal:** Harden the system against "Trojan Horse" attacks and optimize queries.

* **Week 5: Dual-Layer Security Implementation**
  + **RLS Hardening:** Refine RLS policies to test "Partial Access" scenarios (User has doc access, but no CRM access).
  + **Error Boundaries:** Implement React Error Boundaries and TanStack Query error handling within widgets to gracefully display "Access Denied" states.
  + **Server Action Security:** Implement Zod validation middleware for all Server Actions.
* **Week 6: Indexing & Query Optimization**
  + **GIN Indexing:** Apply jsonb\_path\_ops index to the documents.content column.
  + **Query Performance:** Benchmark "Find all docs with CRM widget" using the Generated Column vs. the raw JSONB column.
  + **Search:** Implement basic Full-Text Search using the tsvector generated column.

### Phase 4: Polish, Testing & Handoff (Weeks 7-8)

**Goal:** Finalize UX and prepare for developer handoff.

* **Week 7: Optimistic UI & Polish**
  + **Optimistic Updates:** Implement optimistic UI for document title changes and widget insertion.
  + **Loading States:** Replace generic spinners with skeleton loaders for the Editor and Widgets to reduce Cumulative Layout Shift (CLS).
* **Week 8: Documentation & SOP Freeze**
  + **Developer Docs:** Finalize the "How to create a new Widget" guide.
  + **Code Freeze:** Lock the core architecture.
  + **Manager's Packet:** Delivery of this final report and the SOP prompts.

## 4. Updated Core Documentation

### 4.1 Refined .cursorrules (The AI Constitution)

*Rationale: This file is critical for keeping the AI coder aligned with our specific architectural constraints, specifically regarding barrel files, server actions, and the hybrid data model.*

# Role & Context

You are a Principal Software Architect specializing in Next.js 14 (App Router), Supabase, and Feature-Sliced Design (FSD).

You are building the "Command Center" ERP (Version 1.0).

# Architecture: Feature-Sliced Modular Monolith

* **Strict FSD Structure:** All business logic resides in src/modules/{domain}.
  + src/app: Routing ONLY. No business logic.
  + src/modules: Domain logic (e.g., crm, editor, auth).
  + src/components: Generic UI primitives (shadcn/ui) ONLY.
* **No Barrel Files:** NEVER create index.ts files that re-export entire modules. Import specific components to prevent tree-shaking failures and circular dependencies.
  + BAD: import { Editor } from '@/modules/editor'
  + GOOD: import { Editor } from '@/modules/editor/components/Editor'

# Critical Rules: Next.js 14 & React

* **Client Boundary:**
  + Page shells (page.tsx) are **Server Components** by default.
  + Editor & Widgets are **Client Components** ('use client').
  + NEVER import a Server Component directly into the Editor. Use specific API endpoints or Server Actions to fetch data.
* **Server Actions:**
  + MUST start with 'use server' at the top of the file.
  + MUST include an explicit await checkAuth() call at the start of the function.
  + MUST validate all arguments using Zod schemas.
  + Treat Server Actions as public API endpoints.

# Critical Rules: Supabase & Data

* **The Hybrid Model:**
  + Documents (content column) store JSONB Layouts.
  + Relational Tables store transactional data.
  + NEVER store transactional business data (e.g., Lead Amount) inside the Document JSONB.
* **Row Level Security (RLS):**
  + RLS is the source of truth.
  + Always handle 403 Forbidden errors gracefully in the frontend (Widget Skeleton/Error State).
* **Generated Columns:**
  + Prefer Postgres Generated Columns over Triggers for metadata extraction from JSONB.

# Coding Standards

* **TypeScript:** Strict mode. No any. Use z.infer<typeof Schema> for shared types.
* **Styling:** Tailwind CSS + cn() utility.
* **Async Registry:** The Widget Registry MUST use next/dynamic with { ssr: false }.

# Specific Instructions for BlockNote

* Separate **Schema** (logic/Zod) from **Component** (UI/React).
* Use useBlockNote context to access the editor instance inside widgets.
* Do NOT store heavy transactional data in Block Attributes. Store only configuration (IDs, filters).

### 4.2 Refined version\_1.0.md (Technical Specification)

*Rationale: This specification removes ambiguity regarding the database schema and the widget loading mechanism, explicitly defining the Generated Columns and the Registry contract.*

# Version 1.0 Specification: Command Center ERP

## 1. System Architecture

### 1.1 Directory Structure (Modified FSD)

/src

├── /app # Next.js App Router (Routing Shells)

│ ├── /(auth) # Login/Register Routes

│ ├── /(dashboard) # Protected Routes

│ │ ├── /documents # Document Viewer

│ │ └── /settings # Workspace Settings

│ └── /api # Edge Functions / Webhooks

├── /modules # FEATURE MODULES (Business Logic)

│ ├── /core # Auth, Profiles, Workspace Context

│ ├── /editor # BlockNote Logic, Registry

│ │ ├── /components # Editor.tsx, Toolbar.tsx

│ │ ├── /schema # Zod Schemas for Blocks

│ │ └── registry.ts # The Dynamic Widget Registry

│ └── /crm # Example Business Domain

│ ├── /components # LeadListWidget.tsx

│ ├── /actions # Server Actions (mutateLeads)

│ └── /types # Database Types

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

├── /lib # SINGLETONS (supabase-browser, utils)

└── /types # GLOBAL TYPES (database.types.ts)

## 2. Database Schema (Supabase)

### 2.1 Core Tables

* **workspaces**: Tenancy root.
* **documents**:
  + id: UUID (PK)
  + content: JSONB (The BlockNote state)
  + title: TEXT
  + workspace\_id: UUID (FK)
  + search\_vector: TSVECTOR (Generated Column from content for FTS)
  + widget\_index: TEXT (Generated Column: extracts all block.type from content)

### 2.2 Domain Tables (Example)

* **crm\_leads**:
  + id: UUID
  + status: TEXT
  + value: NUMERIC
  + workspace\_id: UUID

## 3. The Widget Interface

### 3.1 The Registry Contract

The registry must export a strongly-typed map. It acts as the implementation file, preventing circular dependencies.typescript

// src/modules/editor/registry.ts

import dynamic from 'next/dynamic';

export const WIDGET\_REGISTRY = {

'crm-leads': dynamic(() => import('@/modules/crm/components/LeadListWidget'), {

loading: () => ,

ssr: false // Widgets strictly client-side

}),

// Future widgets added here

};

### 3.2 Security Model (Dual-Layer)  
\* \*\*Document Access\*\*: Controls viewing the \*layout\* via `documents` table RLS.  
\* \*\*Widget Access\*\*: Controls viewing the \*data\* via `crm\_leads` table RLS.  
\* \*\*Result\*\*: If User A has Document Access but no CRM Access, they see the document text, but the CRM widget renders a "Permission Denied" placeholder.  
  
## 4. Acceptance Criteria  
1. \*\*Persistence\*\*: Reloading the page restores the exact block structure.  
2. \*\*Isolation\*\*: Adding a widget does not increase the initial bundle size of the Editor (verified via bundle analyzer).  
3. \*\*Performance\*\*: Querying "documents with CRM widgets" uses the SQL index, not JSON parsing.  
4. \*\*Security\*\*: Manually invoking a Server Action for a workspace the user is not part of returns an error.

## 5. The Execution SOP (The Handoff)

This Standard Operating Procedure (SOP) consists of precise "Meta-Prompts" to be fed to the coding agent (Claude 4.5 Opus). These prompts use a "Role-Task-Constraint" framework to ensure adherence to the architectural decisions.

### Phase 1 Prompt: The Skeleton & Hybrid Database

**Objective:** Set up the repo, strict directory structure, and the Supabase schema with Generated Columns.

Prompt:

"Act as a Principal Database Architect and Next.js Expert. We are initializing the 'Command Center' ERP V1.0.

**Task 1: Folder Structure.** Generate a bash script to create the specific directory structure defined in version\_1.0.md. Enforce the 'Modified Feature-Sliced Design' where src/app is for routing only and src/modules is for business logic. Create modules for core, editor, and crm. Remove any default components folder if it exists at the root.

**Task 2: Database Schema (SQL).** Write a comprehensive Supabase SQL migration file.

* Create tables: workspaces, profiles, documents.
* Enable RLS on ALL tables with a default 'deny all' policy.
* **CRITICAL:** Define a GENERATED COLUMN on documents called widget\_index (text array) that extracts the type field from the JSONB content column using jsonb\_path\_query\_array.
* Create a GIN index on the content column using jsonb\_path\_ops for efficient containment queries.

**Task 3: Type Generation.** Explain how to generate the TypeScript types (database.types.ts) from this schema and where to place them in the strict folder structure.

**Constraint:** Do not write React components yet. Focus strictly on the file system and the database layer. Use strictly typed SQL."

### Phase 2 Prompt: The Editor & Registry Implementation

**Objective:** Build the Editor component and the dynamic registry system.

Prompt:

"Act as a Senior React Developer specializing in Performance and Bundle Optimization.

**Context:** We have the database setup. Now we need the frontend Editor Engine.

**Task 1: The Editor Component.** Create src/modules/editor/components/Editor.tsx.

* It must use @blocknote/react.
* It must be a Client Component ('use client').
* Implement a useDebounce hook to auto-save the document JSON to Supabase using a Server Action.

**Task 2: The Registry.** Create src/modules/editor/registry.ts.

* Use next/dynamic to lazy-load a placeholder widget.
* **Constraint:** You must use ssr: false for the dynamic imports to prevent server-side hydration mismatches.
* Define a TypeScript type WidgetKey that locks the allowed widget strings to match the database schema.

**Task 3: Integration.** Show how to mount the Editor component inside src/app/(dashboard)/documents/[id]/page.tsx.

* **Constraint:** The page shell is a Server Component. It must fetch the initial document data and pass it as props to the Client Editor component."

### Phase 3 Prompt: The Secure Data Widget

**Objective:** Build the CRM widget and prove the dual-layer security model.

Prompt:

"Act as a Full-Stack Security Engineer.

**Task 1: The CRM Widget.** Create src/modules/crm/components/LeadListWidget.tsx.

* It should use tanstack-query to fetch leads from Supabase.
* It must handle error states (specifically 403 Forbidden) to render an 'Access Denied' UI if the user lacks table permissions.
* It must NOT import any server-side logic directly.

**Task 2: The Server Action.** Create src/modules/crm/actions/getLeads.ts.

* Use 'use server'.
* Validate the user's session and workspace membership explicitly at the top of the function.
* **Critical:** Show how to enforce RLS context in the Supabase client creation within the action (using createClient with cookies).

**Task 3: BlockNote Integration.** Write the code to register this React component as a custom block in the BlockNote schema.

* Define the Zod schema for the block attributes (e.g., filterStatus).
* Ensure the widget does NOT store the lead data in the JSON attributes, only the configuration."

### Phase 4 Prompt: Performance Optimization

**Objective:** Implement "Generated Columns" query logic and indexing.

Prompt:

"Act as a PostgreSQL Performance Specialist.

**Context:** We have documents containing widgets. We need to efficiently find all documents that contain a 'crm-widget'.

**Task 1: The Query.** Write a Supabase/Postgres query that utilizes the widget\_index Generated Column we created in Phase 1.

* Do NOT query the JSONB blob directly.
* Query the text generated column using the array overlap operator (&&) or containment (@>).

**Task 2: The Index.** Write the SQL to index this generated column for maximum read performance (B-Tree or GIN on the array).

**Task 3: Explanation.** Briefly explain to the junior devs why querying this generated column is faster than documents.content ->> 'type' and how it bypasses the TOAST mechanism."