INVOICE FLOW: The Architectural Blueprint for Next-Generation Desktop Financial Ecosystems

Executive Summary: The Local-First Renaissance

The software industry stands at a pivotal juncture. For the past decade, the dominant paradigm has been cloud-first, browser-based SaaS (Software as a Service). While effective for collaboration, this model has introduced significant latency, privacy concerns, and a homogenization of user interfaces that prioritize utility over experience. We are now witnessing the "Local-First Renaissance," a movement driven by the desire for zero-latency interactions, absolute data sovereignty, and rich, native-feeling user experiences that web browsers alone cannot sustain.

INVOICE FLOW is conceived not merely as a utility for generating financial documents but as a flagship implementation of this new paradigm. By leveraging **Tauri 2.0**, **Rust**, and the latest advances in frontend engineering, INVOICE FLOW bridges the gap between the raw performance of native code and the expressive flexibility of modern web design. The objective is to deliver a desktop application that allows users to construct, customize, and analyze financial documents with a level of fluidity and aesthetic refinement that elicits an immediate "WOW" response.

This report articulates a comprehensive architectural strategy for INVOICE FLOW. It details the technical foundations required to support the user's immediate needs—dynamic invoice generation, structural customization, and local storage—while establishing a modular "Workspace" architecture capable of scaling into a full-fledged Customer Relationship Management (CRM) and Accounting suite. Furthermore, it explores the integration of 2026 design trends, specifically "Corporate Kinetic" aesthetics and Bento Grid dashboards, to redefine the visual standard of enterprise software.

Part I: The Core Architecture – Tauri 2.0 & Rust

The foundation of INVOICE FLOW is the **Tauri 2.0** framework. Unlike its predecessors, such as Electron, which bundle a full instance of the Chromium browser and Node.js runtime with every application—often resulting in binary sizes exceeding 100MB and high memory consumption—Tauri utilizes the operating system's native webview (WebView2 on Windows, WebKit on macOS and Linux). This architectural choice drastically reduces the application's footprint to mere megabytes while enhancing security and performance.

1.1 The Multi-Process Security Model

Central to the architecture of INVOICE FLOW is the strict separation of concerns enforced by Tauri's multi-process model. This model is not just a technical implementation detail but a security and stability guarantee that is essential for financial software.[1]

1.1.1 The Core Process (The Sovereign Backend)

The Core Process serves as the application's central nervous system. Written in **Rust**, it is the only component of the application with direct, unrestricted access to the operating system's resources. In the context of INVOICE FLOW, the Core Process manages the heavy lifting that would typically choke a JavaScript thread.

- **System-Level Orchestration:** The Core creates and manages application windows, system tray integrations, and native menus. It is responsible for the lifecycle of the application, handling startup sequences, update checks, and graceful shutdowns.
- **Data Sovereignty:** All database interactions—reading from the SQLite transactional store or querying the DuckDB analytical engine—occur within the Core. This ensures that the frontend never directly touches the database file, preventing SQL injection attacks or data corruption from a crashed renderer.
- **Cryptographic & File Operations:** Financial documents often require digital signing or secure hashing. The Rust backend handles these cryptographic primitives with native speed. Furthermore, the handling of user assets, such as uploaded company logos or exported PDF archives, is managed here to ensure file system integrity.

1.1.2 The WebView Process (The Ephemeral Frontend)

The WebView Process is the presentation layer. It renders the User Interface (UI) using standard web technologies (HTML, CSS, JavaScript/TypeScript). However, unlike a standard web page, this environment is strictly sandboxed.

- **Isolation Pattern:** The WebView is prohibited from executing system calls directly. It cannot spawn shell processes or read arbitrary files. Instead, it must communicate its intent to the Core Process via a secure Inter-Process Communication (IPC) bridge.
- **Performance Implications:** By offloading logic to the Rust Core, the WebView remains dedicated to rendering. This separation allows INVOICE FLOW to maintain a consistent 60 frames-per-second (FPS) frame rate, ensuring that complex animations—such as the "Kinetic" movement of invoice cards—remain fluid even while the backend performs complex tax calculations on thousands of historical records.[3]

1.2 The Rust Workspace Strategy for Scalability

The requirement to evolve INVOICE FLOW from a simple invoice generator into a CRM and Accounting suite dictates a **Modular Monolith** architecture. A monolithic codebase would quickly become unmanageable, while a microservices architecture introduces unnecessary complexity for a local desktop app. The solution lies in **Rust Workspaces**.

A Workspace allows multiple library crates to coexist within a single repository, sharing a common Cargo.lock file and output directory. This structure enforces modularity at the compiler level, ensuring that distinct business domains remain decoupled.[5]

1.2.1 Directory Structure & Crate Isolation

The proposed directory structure for the INVOICE FLOW monorepo is designed for long-term maintainability:

```
/invoice-flow-monorepo
├── /apps
│ └── /desktop // The Tauri 2.0 Entry Point
│ ├── src-tauri/
│ │ ├── Cargo.toml // Defines the workspace members
│ │ ├── tauri.conf.json // Capabilities & Permissions
│ │ └── src/lib.rs // Plugin Registration & IPC Routing
│ └── src/ // The React 19 Frontend
├── /crates // The Shared Logic Libraries
│ ├── /flow-core // Shared Types, Error Handling, & Traits
│ ├── /flow-db // SQLite Schema, Migrations, & Connection Pooling
│ ├── /flow-analytics // DuckDB Integration for Dashboards
│ ├── /flow-invoice // Business Logic: Tax, Totals, Validation
│ ├── /flow-crm // (Future) Contact Management & Interaction History
│ └── /flow-pdf // Headless Chrome / Print Generation Logic
└── /shared-types // TypeScript definitions generated from Rust structs
```

The flow-core Crate:

This crate acts as the "Standard Library" for the application. It defines the fundamental data types that are shared across the system. For example, the Currency struct, which handles safe decimal math to prevent floating-point errors (e.g., $0.10 + $0.20 = $0.300000004), is defined here. By centralizing these types, we ensure that the Invoice module and the future Accounting module utilize the exact same mathematical standards.

The flow-db Crate:

This crate encapsulates all interactions with the SQLite database. It uses **SQLx**, an async, pure Rust SQL crate that provides compile-time verification of queries. If a developer attempts to select a column that does not exist, the application will fail to compile. This level of safety is critical for financial software, where runtime errors can lead to data loss. This crate exposes high-level functions like create_invoice() or get_client_history(), completely hiding the SQL implementation details from the rest of the app.

The flow-invoice Crate:

This crate contains the pure business logic. It does not know about the database or the UI. It simply accepts data structures (e.g., an InvoiceDraft) and performs operations (e.g., calculate_totals, validate_tax_id). This isolation makes the logic highly testable; we can write thousands of unit tests for edge cases in tax calculation without ever launching the UI.

1.3 Inter-Process Communication (IPC) Design

In Tauri 2.0, the mechanism for communication between the Frontend and the Core has been refined into a robust Command and Event system.

1.3.1 Asynchronous Commands

Frontend actions that require a response utilize **Commands**. For example, when the user clicks "Save Invoice," the React frontend invokes the save_invoice command.

TypeScript

```typescript
// Frontend (React)
import { invoke } from '@tauri-apps/api/core';

async function handleSave() {
 try {
  const result = await invoke('save_invoice', {
   invoice: currentInvoiceData
  });
  console.log("Invoice ID:", result.id);
 } catch (error) {
  console.error("Save failed:", error);
 }
}
```

On the Rust side, this command is an asynchronous function annotated with #[tauri::command]. It runs on a separate thread pool, ensuring the UI does not freeze while the data is being written to the disk.

1.3.2 Real-Time Events

For the dashboard analysis, where data might take a moment to crunch, or for the "Live Preview" of the PDF generation, we utilize **Events**. The Rust Core can emit events like analysis-progress or pdf-ready. The frontend subscribes to these events using listen().

Rust

```
// Backend (Rust)
app.emit("analysis-progress", Payload { percent: 50 }).unwrap();
```

This event-driven architecture is key to the "WOW" factor. It allows the UI to display live progress bars, toast notifications, and real-time status updates, making the application feel responsive and alive.

Part II: Data Persistence – The Hybrid Database Model

A "Stunning Dashboard" for "Complete Analysis" fundamentally changes the data requirements. A simple transactional database is sufficient for storing invoices, but it is often inefficient for complex analytical queries over years of data. Therefore, INVOICE FLOW will implement a **Hybrid Database Architecture**, utilizing **SQLite** for the system of record and **DuckDB** for high-performance analytics.

2.1 SQLite: The Transactional Backbone

SQLite is the de facto standard for local application storage. It is serverless, self-contained, and most importantly, ACID-compliant (Atomicity, Consistency, Isolation, Durability). For INVOICE FLOW, SQLite ensures that financial records are stored reliably.[7]

2.1.1 Schema Design & Migrations

The database schema must be designed to support the dynamic nature of the "User Editable

Structure." We cannot simply have a fixed number of columns for an invoice. Instead, we use a hybrid relational/JSON approach.

- **invoices Table:** Stores the fixed metadata (ID, Invoice Number, Date, Client ID, Total Amount, Status).
- **invoice_items Table:** Stores the line items (Description, Quantity, Rate).
- **invoice_structure Column:** Within the invoices table, a JSON column stores the specific layout configuration used for that invoice. This allows the user to radically change the template for *future* invoices without breaking the rendering of *historical* ones.

To manage schema evolution—for example, adding a crm_id column when the CRM module is released—we utilize **SQLx Migrations**. These are version-controlled SQL files (e.g., 20260218_init.sql, 20260601_add_crm.sql) that run automatically when the app starts, ensuring the user's database is always up to date.

2.1.2 Performance Tuning

Desktop applications require specific SQLite configurations to ensure fluidity:

- **WAL Mode (Write-Ahead Logging):** We enable PRAGMA journal_mode=WAL;. This allows concurrent reading and writing. The dashboard can read data for analysis while the user is simultaneously saving a new invoice, without locking the database.
- **Synchronous Normal:** PRAGMA synchronous=NORMAL;. This setting provides a massive performance boost for write operations while maintaining a high degree of safety against power failure.

2.2 DuckDB: The Analytical Engine

The user's request for "Complete Analysis" and "Stunning Dashboards" is where **DuckDB** enters the architecture. DuckDB is an in-process SQL OLAP database. Unlike SQLite, which processes data row-by-row, DuckDB is **columnar**. This means it is optimized for aggregations—summing columns, calculating averages, and grouping data.[8]

2.2.1 The ETL Pipeline

It is not necessary to store data primarily in DuckDB. Instead, we treat DuckDB as an acceleration layer.

1. **Extraction:** When the user navigates to the "Dashboard" tab, the flow-analytics crate is triggered.
2. **Loading:** DuckDB has the unique ability to read directly from SQLite files. We execute a query in DuckDB: ATTACH 'invoices.db' AS sqlite_db (TYPE SQLITE);.
3. **Transformation:** We can now run complex analytical SQL directly on the SQLite data through the DuckDB engine.
   - *Example Query:* "Calculate the Month-over-Month growth rate of Revenue for the last 36 months, grouped by Client Region."
   - *Performance:* In benchmarks, DuckDB performs these types of queries 10x to 50x

faster than SQLite, enabling real-time interactive graphing on the dashboard without lag.

2.2.2 Future-Proofing for Accounts

As the application expands into Accounting, DuckDB becomes even more critical. It can handle the "Double-Entry Ledger" calculations—summing millions of debit and credit entries to produce a Balance Sheet or Profit & Loss statement—in milliseconds. This capability effectively future-proofs the application's performance characteristics against massive dataset growth.

Part III: Frontend Engineering – The Editor Engine

The frontend is where the user spends their time. To achieve the "WOW" factor, we must move beyond standard HTML forms. The User Experience (UX) goal is a "WYSIWYG" (What You See Is What You Get) document editor that feels indistinguishable from the final PDF.

3.1 Tech Stack: React 19 & Vite

We utilize **React 19** to leverage its concurrent rendering features. This ensures that the UI remains responsive even during heavy rendering updates, such as dragging complex invoice blocks. **Vite** is the build tool of choice, providing an instant Hot Module Replacement (HMR) workflow that accelerates development.[10]

3.2 The Dynamic Block Engine

The requirement "User's able to edit the structure" necessitates a **Block-Based Editor Architecture**, conceptually similar to Notion or WordPress Gutenberg but specialized for invoices.[11]

3.2.1 The Data Model

Instead of a flat object, the invoice state is represented as a list of **Blocks**.

JSON

```json
{
 "meta": { "id": "INV-1001", "date": "2026-10-15" },
 "layout": [
  {
    "id": "block_1",
    "type": "header",
    "settings": { "align": "left", "show_logo": true }
  },
  {
    "id": "block_2",
    "type": "grid_container",
    "columns": [
      { "type": "bill_to_client" },
      { "type": "project_meta" }
    ]
  },
  {
    "id": "block_3",
    "type": "services_table",
    "settings": { "columns": ["desc", "qty", "rate", "total"] }
  }
 ]
}
```

3.2.2 Drag-and-Drop Implementation (dnd-kit)

To allow users to reorder these sections (e.g., moving "Project Details" above "Bill To"), we utilize **dnd-kit**. It is a modern, lightweight drag-and-drop library for React that supports accessibility and touch inputs.[13]

- **Sortable Context:** The main document container is wrapped in a <SortableContext>. Each block (Header, Table, Footer) is a sortable item.
- **Drag Overlay:** When a user grabs a block, dnd-kit creates a "Drag Overlay"—a semi-transparent ghost of the block that follows the cursor.
- **Physics & Animation:** We use **Framer Motion** for the reordering animations. As the user drags a block, the other blocks gently slide out of the way using a spring physics simulation, creating a tactile, high-quality feel.

3.3 The "Invisible" Input System

To achieve the "Clean, minimal" aesthetic, we must banish standard input boxes with gray borders. The interface should look like the final document.

- **Inline Editing:** We utilize "Content Editable" components. A text field looks like plain text. When clicked, it gains a subtle caret. There are no jarring transitions between "Read Mode" and "Edit Mode."
- **Contextual Controls:** Controls for a block (e.g., "Add Column," "Change Color") appear only when the mouse hovers over that specific block. This keeps the interface noise-free until the user intends to edit.

3.4 The Calculation Engine (Zustand)

Financial applications demand immediate feedback. If a user changes a quantity, the Grand Total must update instantly.

- **State Management:** We use **Zustand** for managing the invoice state. It is lighter than Redux and allows for transient updates (high-frequency changes) without re-rendering the entire component tree.
- **Reactive Logic:** We create "Selectors" that automatically derive values.

  ```TypeScript
  const useSubtotal = () => useInvoiceStore(state =>
   state.items.reduce((sum, item) => sum + (item.qty * item.rate), 0)
  );
  ```

- **Floating Point Safety:** JavaScript's native math is notorious for precision errors. We will use a library like decimal.js or currency.js within the frontend to ensure that $19.99 * 3 results in $59.97, not $59.970000000004.

Part IV: Visual Design System – "Corporate Kinetic"

To elicit the "WOW" response, the design must strike a delicate balance. It must be "Corporate" (trustworthy, stable, legible) yet "Kinetic" (fluid, responsive, modern).

4.1 The Palette: Trust & Energy

The color strategy leverages the psychology of color in finance.[15]

- **The Foundation (Navy):** Midnight Navy (#0A192F) and Deep Slate (#172A45) form the background of the application shell. Dark blue is universally associated with stability, banking, and authority.
- **The Canvas (White/Paper):** The invoice area itself is Pure White (#FFFFFF) or Ghost White (#F8FAFC). This creates a high contrast with the dark app shell, focusing the user's attention strictly on the document.
- **The Accent (Teal/Coral):** To prevent the app from feeling boring, we use vibrant accents. Electric Teal (#64FFDA) is used for positive actions (Save, Send, Paid). Soft Coral (#EE6D66) provides a warm contrast for alerts or "Pay Now" buttons on the invoice itself.

4.2 The Dashboard: The Bento Grid Layout

The dashboard is the first thing the user sees. It must not be a spreadsheet. It must be a cockpit. We will implement a **Bento Grid** layout.[16]

- **Modular Tiles:** The dashboard is composed of rectangular tiles of varying sizes (1x1, 2x1, 2x2). This allows for a flexible arrangement of information density.

- **Information Hierarchy:**
  - *The Hero Tile (2x2):* "Total Revenue." This is the most critical metric. It features a large, bold number and a background sparkline chart.
  - *The Status Tile (1x2):* "Outstanding Invoices." A circular gauge showing the ratio of paid vs. unpaid.
  - *The Action Tile (1x1):* "Create New." A purely functional tile with a large "+" icon.
- **Psychological Impact:** The Bento Grid reduces cognitive load (Hick's Law) by compartmentalizing information into digestible, self-contained chunks. The irregular grid pattern is visually more engaging than a list, encouraging exploration.

4.3 Motion Design & Micro-Interactions

Static interfaces feel dated. "Corporate Kinetic" implies that the interface responds to the user's presence.

- **3D Card Effects:** Using **Aceternity UI** components, dashboard cards can have a subtle 3D tilt effect on hover. As the mouse moves over a card, it rotates slightly in 3D space, following the cursor. This adds a sense of depth and tangibility to the digital objects.[17]
- **Number Rolling:** When data updates (e.g., after saving a new invoice), the "Total Revenue" number shouldn't just swap. It should "roll" or "count up" to the new value (like a mechanical odometer). This draws attention to the change and creates a feeling of accumulation.
- **Fluid Layouts:** When the window is resized, the Bento Grid tiles should not just snap. They should use **Framer Motion** layout animations to glide into their new positions, maintaining spatial relationships.

Part V: Functional Requirements Deep Dive

This section details the specific implementation of the user's requested features within the architectural framework.

5.1 The Invoice Structure Components

1. **Header Section (Sender & Meta):**
   - *Implementation:* A Flexbox container allowing left/center/right alignment.
   - *Logo:* The logo is fetched from the local asset store. We implement a "Smart Color Extraction" feature (using a library like colorthief) to automatically suggest a "Primary Color" for headers that matches the user's uploaded logo.
2. **Bill To Section (Client Details):**
   - *Design:* A card with a "Colored Border Accent" on the left. This accent color is user-customizable or derived from the logo.
   - *Interaction:* This is the integration point for the future CRM. Currently, it is an auto-complete text area. Later, it will pull from the contacts table.

3. **Project Details (Key-Value Grid):**
   - *Structure:* A CSS Grid layout (grid-cols-2).
   - *Dynamic Keys:* The user can add arbitrary rows (e.g., "PO Number," "Tech Stack," "Hosting"). This maps to a JSON object in the database, preserving flexibility.
4. **Services & Charges Table:**
   - *Columns:* Description, Quantity, Rate, Amount.
   - *UX:* The "Totals Row" uses a larger font weight (font-bold) and a background tint to visually separate it from the line items.
   - *Drag-to-Reorder:* Users can drag rows to re-prioritize items.
5. **Payment Information (Bank & QR):**
   - *Layout:* A side-by-side grid. Left side: Bank Text. Right side: Dynamic QR Code.
   - *QR Logic:* We use the qrcode.react library. The QR code is generated locally in the client. It can encode a standard URL, a Bitcoin address, or a SEPA/UPI payment string.
6. **Payment Terms:**
   - *Design:* A "Soft-colored box." We use Tailwind's bg-opacity utilities (e.g., bg-blue-500/10) to create a subtle, non-intrusive container that highlights the policy without screaming for attention.
7. **Footer (Disclaimer):**
   - *Logic:* A computed text string. "Computer Generated Invoice. No Signature Required."
   - *Print Behavior:* Using CSS @media print, we force this element to the bottom of the page (position: fixed; bottom: 0), ensuring it always acts as a formal footer.

5.2 Logo Upload & Asset Management

Handling local images requires navigating the browser's security model.

- **Storage:** When a logo is uploaded, the Core Process hashes the file (SHA-256) to detect duplicates and saves it to $APPDATA/InvoiceFlow/assets/logos/.
- **Retrieval:** We configure Tauri's tauri.conf.json to enable the **Asset Protocol**.
  ```json
  JSON
  "app": {
   "security": {
    "assetProtocol": {
     "scope":
    }
   }
  }
  ```
  The frontend then requests the image via a secure URL: asset://localhost/$APPDATA/.... This bypasses the browser's restrictions on loading local file resources.[19]

5.3 Auto-Increment Invoice Numbers

To prevent duplicates and ensure sequential numbering, this logic lives strictly in the Rust Core

/ SQLite layer.

- **Mechanism:** When a user initiates "New Invoice," the frontend requests a draft.
- **Concurrency Safety:** The Rust backend opens a transaction. It queries SELECT MAX(sequence_number) FROM invoices WHERE year = 2026. It increments the result by one and reserves this number.
- **Format:** The user can define a format string (e.g., INV-{YYYY}-{000}). The backend parses this string and injects the reserved sequence number.

Part VI: The Output Engine – PDF & Printing

The user specifically requested "Export PDF (A4 formatting via @media print)." However, achieving pixel-perfect A4 results across different operating systems using only CSS is notoriously difficult due to varying browser rendering engines and default print margins. We propose a **Dual-Strategy** to ensure reliability.

6.1 Strategy A: CSS Print Media (The Native approach)

This satisfies the explicit requirement and is fast for quick prints.

- **The @page Rule:** We define the page size explicitly in CSS.
  CSS
  ```css
  @page {
   size: A4 portrait;
   margin: 0; /* Critical: We handle margins inside the container */
  }
  ```

- **The Paper Container:** We create a div that mimics the physical paper.

.invoice-sheet {

width: 210mm;

min-height: 297mm; /* Exact A4 height / *padding: 20mm;* / The actual visual margin */

background: white;

box-sizing: border-box;

overflow: hidden;

}

```

- **Print Specifics:** Inside @media print, we hide all UI chrome (sidebars, buttons) and ensure the background colors are forced to render (-webkit-print-color-adjust: exact).
```

6.2 Strategy B: Headless Chrome Rendering (The "Pro" Export)

For "Export to PDF" where the file must look *exactly* the same on a Mac as it does on Windows, relying on the user's print driver is risky. We implement a backend generator.

- **Mechanism:** The Rust Core spawns a hidden, headless WebView instance (or uses the headless_chrome crate).
- **Process:**
  1. The frontend sends the Invoice HTML/JSON to the backend.
  2. The backend injects this data into a sanitized Print Template.
  3. The headless browser renders this template and uses its internal "Print to PDF" API.
  4. The binary PDF data is streamed back to the Core, which saves it to the user's disk.
- **Benefit:** This guarantees that fonts, line heights, and page breaks are mathematically identical every time, regardless of the user's printer settings.[21]

Part VII: Future Scalability – CRM, Accounts, & AI

The architecture is designed to accommodate the user's future roadmap without rewriting the core.

7.1 The CRM Plugin

Using the workspace architecture, the CRM will be a separate crate (flow-crm).

- **Integration:** The Invoice "Bill To" block will upgrade to a "Smart Search" block. It will query the CRM crate via IPC.
- **Data Link:** The invoices table will gain a client_id foreign key linking to the clients table in the CRM module. This enables queries like "Show total revenue for Client X."

7.2 The Accounting Module & AI Vibe Coding

- **Double-Entry Engine:** The flow-accounts crate will implement a double-entry ledger. Every invoice generated will automatically create a "Credit" entry in "Sales" and a "Debit" entry in "Accounts Receivable."
- **AI Integration:** We can integrate a small, local Large Language Model (LLM) like Llama-3-8B (quantized) or use cloud APIs if permitted.
  - *Feature: "Smart Categorization."* When a user adds an expense line item "AWS Hosting," the AI analyzes the text and automatically suggests the "Infrastructure" ledger account.
  - *Feature: "Natural Language Analytics."* The user can type "How much did I make last November?" into the dashboard search bar. The AI translates this into a DuckDB SQL query and renders the result. This aligns with the "Vibe Coding" trend where software anticipates intent.[23]

Conclusion

INVOICE FLOW represents the convergence of three powerful trends: the efficiency of **Local-First** software, the performance of **Rust/Tauri 2.0**, and the aesthetic elevation of **Corporate Kinetic** design.

By adhering to the architectural blueprint laid out in this report—specifically the use of Rust Workspaces for modularity, a Hybrid SQLite/DuckDB data layer for analytical power, and a dnd-kit based Block Editor for structural flexibility—developers can build a system that meets the user's immediate needs while robustly supporting future expansion. The result will not just be an invoicing tool, but a piece of "WOW" software that redefines what users expect from their financial utilities.

Works cited

1. Tauri 2.0: The Unified Frontier of Cross-Platform Development | Uplatz - YouTube, accessed February 18, 2026, https://www.youtube.com/watch?v=hYntdmO-HvQ
2. Process Model - Tauri, accessed February 18, 2026, https://v2.tauri.app/concept/process-model/
3. Why I chose Tauri - Practical advice on picking the right Rust GUI solution for you - Reddit, accessed February 18, 2026, https://www.reddit.com/r/rust/comments/1ihv7y9/why_i_chose_tauri_practical_advice_on_picking_the/
4. Built a desktop app with Tauri 2.0 - impressions after 6 months : r/rust - Reddit, accessed February 18, 2026, https://www.reddit.com/r/rust/comments/1nvvoee/built_a_desktop_app_with_tauri_20_impressions/
5. Is there a way to integrate Tauri 2 and Rust workspaces? - Reddit, accessed February 18, 2026, https://www.reddit.com/r/tauri/comments/1jr611u/is_there_a_way_to_integrate_tauri_2_and_rust/
6. Rust Workspace Example: A Guide to Managing Multi-Crate Projects | by UATeam - Medium, accessed February 18, 2026, https://medium.com/@aleksej.gudkov/rust-workspace-example-a-guide-to-managing-multi-crate-projects-82d318409260
7. DuckDB vs SQLite: A Complete Database Comparison - DataCamp, accessed February 18, 2026, https://www.datacamp.com/blog/duckdb-vs-sqlite-complete-database-comparison
8. DuckDB vs SQLite: Which Embedded Database Should You Use? - MotherDuck, accessed February 18, 2026, https://motherduck.com/learn-more/duckdb-vs-sqlite-databases/
9. DuckDB vs SQLite: Performance, Speed, and Use Cases Compared, accessed February 18, 2026, https://www.hakunamatatatech.com/our-resources/blog/sqlite
10. What is Tauri?, accessed February 18, 2026, https://v2.tauri.app/start/

11. BlockNote - Javascript Block-Based React rich text editor, accessed February 18, 2026, https://www.blocknotejs.org/
12. BlockNote Review: The Ultimate Open Source Block Editor (2025) - Velt, accessed February 18, 2026, https://velt.dev/blog/blocknote-collaborative-editor-guide
13. React Drag And Drop (dnd-kit) | Beginners Tutorial - YouTube, accessed February 18, 2026, https://www.youtube.com/watch?v=dL5SOdgMbRY
14. I Built a Drag-and-Drop React Form Builder with Zod & React Hook Form – Here's How, accessed February 18, 2026, https://dev.to/shoaeeb_osman_e3e2ae43910/i-built-a-drag-and-drop-react-form-builder-with-zod-react-hook-form-heres-how-3bc4
15. 20 Dark Blue Color Palette Combinations & Hex Codes - Media.io, accessed February 18, 2026, https://www.media.io/color-palette/dark-blue-color-palette.html
16. Bento Grid Design: How to Create Modern Modular Layouts in 2026 - Landdding, accessed February 18, 2026, https://landdding.com/blog/blog-bento-grid-design-guide
17. Free React & Next.js Components — Tailwind CSS & Framer Motion ..., accessed February 18, 2026, https://ui.aceternity.com/components
18. Top 10 UI library to skyrocket your Frontend in 2026! | Nyxhora Blog, accessed February 18, 2026, https://www.nyxhora.com/blog/top-10-ui-library-to-skyrocket-your-frontend-in-2026-8923
19. How to load local resources : r/tauri - Reddit, accessed February 18, 2026, https://www.reddit.com/r/tauri/comments/1js8n9t/how_to_load_local_resources/
20. File System - Tauri, accessed February 18, 2026, https://v2.tauri.app/plugin/file-system/
21. tauri-plugin-printer-wkhtml-bin - crates.io: Rust Package Registry, accessed February 18, 2026, https://crates.io/crates/tauri-plugin-printer-wkhtml-bin/0.1.0
22. Making a pdf printing tool: need tech stack advice : r/rust - Reddit, accessed February 18, 2026, https://www.reddit.com/r/rust/comments/194hc4z/making_a_pdf_printing_tool_need_tech_stack_advice/
23. accessed February 18, 2026, https://juliangoldie.com/google-stitch-skills-in-antigravity/#:~:text=Stitch%20is%20a%20front%2Dend,ships%20UI%20projects%20for%20you.
24. Google Stitch and AI Studio Workflow — Build Real Apps Without Code - Reddit, accessed February 18, 2026, https://www.reddit.com/r/AISEOInsider/comments/1qgeq1d/google_stitch_and_ai_studio_workflow_build_real/