Extensible BI Dashboard Framework - Technical Documentation

Jehu Shalom Amanna - Frontend Solutions Architect

October 27, 2025

Abstract

Extensible BI Dashboard Framework is a browser-based, highly extensible framework designed for building dynamic Business Intelligence (BI) dashboards. The system features a minimal core with maximum extensibility, allowing developers to create, customize, and extend UI components on-the-fly using both a custom DSL and JavaScript.

Core Philosophy:

- **Tiny Core, Maximum Extensibility**: Minimal core functionality with comprehensive plugin architecture
- **Plugin-Based Architecture**: Adopts proven patterns for extensibility and modularity
- **Developer-First Design**: Prioritizes developer experience with hot reloading, debugging tools, and clear APIs

Contents

Chapter 1

Architecture Diagram

1.1 Architecture (reflects comprehensive system design):

Layer 1: User Interface Layer

Dashboard Builder	**Command Palette**	**Extension Manager**	**Settings Panel**	**Theme Switcher**
Layout Manager	**Keybinding Editor**	**Hooks Inspector**	**Advice Debugger**	

↓

Layer 2: Core System Layer

Component Registry (Lazy Load)	**Event System** (Typed Events)	**State Management** (Zustand)
Plugin Loader (HMR + DI)	**Keybinding System** (Chord + Ctx)	**Command Registry** (Palette)
Hooks & Advice (Priority)	**Theme System** (CSS Vars)	**Layout Engine** (Grid/Mosaic)

↓

Layer 3: Extension Layer

DSL Extensions	**JavaScript Extensions**	**React Components**	**Web Components**	**Themes & Layouts**
Commands	**Keybindings**	**Hooks**	**Macros**	

↓

Layer 4: Security Layer

Sandboxed Execution (SES/iframe)	**Capability Permissions** (Runtime)	**Code Signing** (Crypto API)
API Surface (Versioned)	**Audit Log** (Tracking)	**Marketplace Review**

↓

Layer 5: Persistence Layer

IndexedDB (Warm Data)	**LocalStorage** (Hot Data)	**OPFS** (Cold Data)
Cloud Sync (REST/GQL)	**DuckDB WASM** (Analytics)	**Time-Travel** (Zundo)

↓

Layer 6: Development Layer

HMR (Vite/WP)	**Source Maps** (Debugging)	**Error Overlay** (Dev Mode)
State Inspector (DevTools)	**Performance Profiling**	**Network Monitoring** (Extension)

1.2 Key Architecture Components

1. **Core System Layer**:
   - Added Hooks & Advice system for extensibility
   - Theme System with CSS Variables
   - Layout Engine (Grid/Mosaic patterns)
   - Enhanced state management (Zustand with middleware)
   - Typed event system with history/replay
2. **Extension Layer**:
   - Expanded to include Web Components
   - Commands and Keybindings as first-class extensions
   - Hooks and Macros support
   - Theme and Layout templates
3. **Security Layer**:
   - Multiple sandboxing approaches (SES/iframe)
   - Runtime permission validation
   - Versioned API surface
   - Audit logging system
   - Marketplace review process
4. **Persistence Layer**:
   - Tiered storage strategy (Hot/Warm/Cold)
   - DuckDB WASM for analytics
   - Time-travel debugging (Zundo)
   - OPFS for large file storage
5. **Development Layer** (New):
   - Hot Module Replacement
   - Source maps and debugging tools
   - Error overlay and recovery
   - State inspector
   - Performance profiling
   - Network monitoring

1.3 Data Flow

```
User Action → UI Layer → Core System → Extension Layer
                ↓             ↓             ↓
```

```
Security Check → Permission → Sandboxed Execution
                      ↓
               Persistence Layer
                      ↓
               Development Tools (Dev Mode)
```

Chapter 2

Core System Architecture

The Core System Architecture represents Layer 2 of the framework, providing the minimal yet comprehensive foundation for the extensible BI Dashboard ecosystem. This layer implements the "Tiny Core, Maximum Extensibility" philosophy by offering essential services that all other layers depend upon: Component Registry for dynamic UI management, Event System for decoupled communication, State Management for reactive data flow, Plugin Loader for extension lifecycle, Keybinding System for user interactions, Command Registry for action orchestration, Hooks & Advice for aspect-oriented programming, Theme System for visual customization, and Layout Engine for flexible dashboard arrangements. Each component is designed to be lightweight, performant, and extensible, enabling plugins and extensions to build upon a solid, predictable foundation without introducing unnecessary complexity or bloat. This architecture ensures that the framework remains fast and maintainable while supporting unlimited extensibility through well-defined interfaces and patterns.

2.1 Component Registry

2.1.1 Purpose

Central registry for all UI components (built-in and user-defined) in the BI Dashboard Framework. The Component Registry acts as the foundation for the extensible architecture, managing the entire lifecycle of visualization components, dashboard widgets, data displays, input controls, and custom extensions. It enables dynamic component discovery and loading, supports hot module replacement for development, handles version compatibility between components, and provides a unified interface for both core framework components and third-party extensions. This registry is essential for the plugin system, allowing extensions to contribute new chart types, data widgets, and UI elements that seamlessly integrate with the dashboard ecosystem.

2.1.2 Core Responsibilities

- Component registration and discovery
- Lifecycle management (mount, unmount, update)
- Dependency resolution
- Version management

2.1.3 Architecture Patterns

Pattern	Description	Pros	Cons	Best For
Service Locator	Central registry with get/set	Simple; Centralized; Easy lookup	Global state; Hidden dependencies; Testing harder	Simple apps, quick prototypes
Dependency Injection	Components receive dependencies	Explicit dependencies; Testable; Decoupled	More boilerplate; Complex setup; Learning curve	Large apps, testability important
Module Federation	Webpack 5 feature for runtime loading	True code splitting; Independent deployment; Version isolation	Webpack-specific; Complex config; Build complexity	Micro-frontends, large teams
Dynamic Import	ES modules with import()	Native support; Code splitting; Simple	Limited metadata; No version control; Manual registry	Modern apps, simple plugins

2.1.4 Library Comparison

Library	Type	Pros	Cons	Bundle Size	Use Case
InversifyJS	DI container	Full DI support; Decorators; TypeScript; Mature	Large bundle; Reflect metadata; Complex API	~15KB	Enterprise apps, complex DI
TSyringe	DI container	Lightweight DI; Decorators; Simple API; TypeScript	Requires decorators; Less features	~3KB	TypeScript apps, moderate DI
Awilix	DI container	No decorators needed; Flexible; Good docs	Less type-safe; Manual setup	~5KB	Node.js-style, flexible DI
Custom Registry	DIY Map/Object	Full control; Minimal size; Simple	Manual implementation; No DI features	<1KB	Simple needs, full control

2.1.5 BI Dashboard Examples

Platform	Registry Pattern	Extension Method
Observable	Module-based registry	• Notebook cells as components; • Dynamic import for modules; • Runtime dependency resolution; • Version pinning per notebook
Evidence	File-based convention	• Components auto-discovered from /components; • Svelte component registry; • Build-time registration; • No runtime DI
Count.co	Component library	• Pre-built visualization components; • SQL-driven component binding; • Canvas-based layout registry; • Drag-and-drop component system
tldraw	Shape registry	• Shape definitions as components; • Tool registry pattern; • Custom shape API; • Runtime shape registration
Omni Docs	Plugin registry	• Plugin-based documentation system; • Markdown-based content; • Custom plugin API; • Runtime plugin registration

2.1.6 Recommended Architecture

```
// Component registry with versioning and lifecycle
interface ComponentMetadata {
  id: string;
  name: string;
  version: string;
  dependencies?: string[];
  lazy?: boolean;
  loader?: () => Promise<Component>;
}

class ComponentRegistry {
  private components = new Map<string, ComponentMetadata>();
  private instances = new Map<string, Component>();

  register(metadata: ComponentMetadata): void {
    // Version conflict check
    if (this.components.has(metadata.id)) {
      const existing = this.components.get(metadata.id)!;
      if (existing.version !== metadata.version) {
        console.warn(`Version conflict: ${metadata.id}`);
      }
    }

    this.components.set(metadata.id, metadata);
  }

  async get(id: string): Promise<Component> {
```

```
    // Check cache
    if (this.instances.has(id)) {
      return this.instances.get(id)!;
    }

    const metadata = this.components.get(id);
    if (!metadata) {
      throw new Error(`Component not found: ${id}`);
    }

    // Resolve dependencies
    if (metadata.dependencies) {
      await Promise.all(
        metadata.dependencies.map(dep => this.get(dep))
      );
    }

    // Lazy load if needed
    const component = metadata.lazy && metadata.loader
      ? await metadata.loader()
      : metadata;

    this.instances.set(id, component);
    return component;
  }

  unregister(id: string): void {
    this.components.delete(id);
    this.instances.delete(id);
  }
}
```

For detailed state management integration, see Section 1.3. For plugin lifecycle, see Section 1.4.

2.2 Event System

2.2.1 Purpose

Pub/sub event bus for inter-component communication in the BI Dashboard Framework. The Event System serves as the nervous system of the application, enabling loosely-coupled communication between components, plugins, and core systems without creating direct dependencies. It facilitates real-time updates across dashboard cells, synchronizes state changes between visualizations, broadcasts user actions to interested listeners, coordinates plugin lifecycle events, and enables reactive data flows throughout the application. This decoupled architecture allows components to communicate efficiently while maintaining modularity—when a data query completes, visualization updates, user interaction occurs, or plugin state changes, relevant components can respond without tight coupling. The system supports both synchronous and asynchronous event handling, provides event history for debugging and time-travel, and offers scoped channels to prevent event namespace pollution in complex dashboards.

2.2.2 Core Features

- Global and scoped event channels
- Event hooks and listeners

- Async event handling
- Event history and replay (for debugging)

2.2.3 Architecture Patterns

Pattern	Description	Pros	Cons	Best For
Event Emitter	Simple pub/sub	Simple; Familiar; Small	No type safety; Memory leaks risk; No scoping	Simple events, small apps
Event Bus	Centralized event hub	Decoupled; Global access; Easy debugging	Global state; Hidden dependencies; Testing harder	Cross-component communication
Observable Streams	RxJS-style	Powerful operators; Composable; Async-friendly	Learning curve; Large bundle; Overkill for simple cases	Complex async flows
Custom Events	DOM CustomEvent	Native API; No dependencies; Bubbling support	DOM-only; Limited features; Verbose	DOM-centric apps

2.2.4 Library Comparison

Library	Type	Pros	Cons	Bundle Size	Use Case
mitt	Event emitter	Tiny (200B); TypeScript; Simple API	Basic features; No scoping; No async	200B	Minimal apps, simple events
eventemitter3	Event emitter	Fast; Mature; Well-tested	No TypeScript; Larger bundle	~2KB	Performance-critical
RxJS	Reactive streams	Very powerful; Operators; Async handling	Large (40KB+); Steep curve; Complex	~40KB	Complex async, data streams
Nano Events	Event emitter	Very small; Simple; TypeScript	Minimal features	200B	Size-constrained
EventEmitter2	Enhanced emitter	Wildcards; Namespaces; Feature-rich	Larger; More complex	~5KB	Complex event patterns

2.2.5 Event System Patterns

Feature	Implementation	Pros	Cons
Event Namespacing	`user:login`, `data:update`	Organization; Wildcards	String-based; No type safety
Typed Events	TypeScript discriminated unions	Type-safe; Autocomplete	More boilerplate; TS-only
Event Replay	Store event history	Debugging; Time-travel	Memory usage; Complexity
Scoped Channels	Separate buses per scope	Isolation; Less noise	More instances; Coordination

2.2.6 BI Dashboard Examples

Platform	Event Pattern	Implementation
Observable	Reactive cells	• Cell dependencies as events; • Automatic re-execution; • Dataflow graph; • No explicit pub/sub
Evidence	Component events	• Svelte component events; • Custom events for data updates; • Build-time event binding
Count.co	Canvas events	• Cell update events; • Query execution events; • Collaboration events (real-time); • Canvas state changes
tldraw	Shape events	• Shape change events; • Selection events; • Canvas interaction events; • History events (undo/redo)
Omni Docs	Plugin events	• Plugin-based event system; • Custom event API; • Runtime event registration

2.2.7 Recommended Architecture

```
// Type-safe event system
type EventMap = {
  'dashboard:loaded': { id: string; data: any };
  'panel:updated': { panelId: string; changes: any };
  'data:fetched': { query: string; result: any };
  'user:action': { action: string; payload: any };
};

class TypedEventBus {
  private emitter = new EventEmitter();
  private history: Array<{ event: string; data: any; timestamp: number }> = [];
  private maxHistory = 100;

  on<K extends keyof EventMap>(
```

```
    event: K,
    handler: (data: EventMap[K]) => void
  ): () => void {
    this.emitter.on(event, handler);
    return () => this.emitter.off(event, handler);
  }

  emit<K extends keyof EventMap>(event: K, data: EventMap[K]): void {
    // Store in history
    this.history.push({ event, data, timestamp: Date.now() });
    if (this.history.length > this.maxHistory) {
      this.history.shift();
    }

    this.emitter.emit(event, data);
  }

  replay(fromTimestamp?: number): void {
    const events = fromTimestamp
      ? this.history.filter(e => e.timestamp >= fromTimestamp)
      : this.history;

    events.forEach(({ event, data }) => {
      this.emitter.emit(event, data);
    });
  }

  getHistory(): typeof this.history {
    return [...this.history];
  }
}
```

For integration with hooks, see Section 2.2.6. For state synchronization, see Section 1.3.

2.3 State Management

2.3.1 Purpose

Centralized, reactive state management for the BI Dashboard Framework. The state management layer serves as the single source of truth for application data, managing dashboard configurations, user preferences, plugin state, visualization data, and UI state. It provides predictable state updates through immutable patterns, enables real-time reactivity across components, and supports advanced features like time-travel debugging, state persistence, and collaborative editing. This layer ensures that all components—from the canvas and cells to plugins and extensions—stay synchronized and can efficiently respond to data changes without prop drilling or excessive re-renders.

2.3.2 Core Features

- Immutable state updates
- Time-travel debugging
- State persistence and hydration
- Computed/derived state
- State snapshots and restoration

2.3.3 Architecture Patterns

Pattern	Description	Pros	Cons	Best For
Flux/Redux	Unidirectional data flow	Predictable; Time-travel; DevTools	Boilerplate; Learning curve; Verbose	Large apps, complex state
Atomic State	Fine-grained atoms	Minimal re-renders; Composable; Simple	Many atoms; Coordination; Less structure	React apps, performance-critical
Proxy-Based	Mutable API with tracking	Simple API; Auto-tracking; Intuitive	Proxy overhead; Debugging harder	Rapid development
Observable	RxJS/MobX style	Reactive; Powerful; Composable	Learning curve; Large bundle	Complex reactive flows

2.3.4 Library Comparison

Library	Pattern	Pros	Cons	Bundle Size	Use Case
Zustand	Flux-like	Simple API; No providers; Middleware; Small bundle	Manual optimization; Less structure	~1KB	General-purpose, React
Jotai	Atomic	Minimal re-renders; Bottom-up; TypeScript; Suspense	Many atoms; Boilerplate; Debugging	~3KB	Performance-critical React
Valtio	Proxy	Mutable API; Auto-tracking; Simple; Snapshots	Proxy limitations; Less predictable	~3KB	Rapid development, simple state
Redux Toolkit	Redux	Less boilerplate; DevTools; Mature; Ecosystem	Still verbose; Learning curve; Larger	~10KB	Enterprise, complex workflows

Library	Pattern	Pros	Cons	Bundle Size	Use Case
MobX	Observable	Very reactive; Automatic tracking; Powerful	Large bundle; Magic behavior; Learning curve	~16KB	Complex reactive apps
Recoil	Atomic	React-first; Async support; Selectors	Experimental; React-only; Less mature	~14KB	React apps, async state
XState	State machines	Predictable; Visualizable; Complex flows	Learning curve; Verbose; Overkill for simple	~10KB	Complex state machines
Signia	Signals	Fine-grained reactivity; track() API; Fast; Framework-agnostic	New library; Smaller ecosystem; tldraw-specific	~5KB	Canvas apps, fine-grained updates

2.3.5 State Management Features

Feature	Implementation	Pros	Cons
Time-Travel	Store action history	Debugging; Undo/redo	Memory usage; Complexity
Persistence	LocalStorage/IndexedDB sync	Survives refresh; User experience	Serialization; Migration
Computed State	Derived values/selectors	DRY principle; Performance	Memoization needed; Complexity
Middleware	Intercept actions	Logging; Analytics; Side effects	Indirection; Debugging

2.3.6 BI Dashboard Examples

Platform	State Pattern	Implementation
Observable	Reactive cells	• Each cell is state; • Automatic dependency tracking; • Dataflow graph execution; • No central store

Platform	State Pattern	Implementation
Evidence	Svelte stores	• Writable stores for state; • Derived stores for computed; • Context for component state
Count.co	Canvas state	• Canvas-level state management; • Cell state with SQL results; • Collaborative state sync; • Local + server state
tldraw	Signia (signals)	• Fine-grained reactive signals; • Shape state management; • History state (undo/redo); • track() for reactive components
Omni Docs	Plugin state	• Plugin-based state management; • Custom state API; • Runtime state registration

2.3.7 Recommended Architecture

```
// Zustand store with persistence and time-travel
import create from 'zustand';
import { persist, devtools } from 'zustand/middleware';
import { temporal } from 'zundo';

interface DashboardState {
  dashboards: Dashboard[];
  activeDashboard: string | null;
  panels: Record<string, Panel>;

  // Actions
  addDashboard: (dashboard: Dashboard) => void;
  updatePanel: (id: string, updates: Partial<Panel>) => void;
  setActiveDashboard: (id: string) => void;

  // Computed (via selectors)
  getActivePanels: () => Panel[];
}

const useDashboardStore = create<DashboardState>()(
  devtools(
    persist(
      temporal(
        (set, get) => ({
          dashboards: [],
          activeDashboard: null,
          panels: {},

          addDashboard: (dashboard) =>
            set((state) => ({
              dashboards: [...state.dashboards, dashboard]
            })),
```

```
          updatePanel: (id, updates) =>
            set((state) => ({
              panels: {
                ...state.panels,
                [id]: { ...state.panels[id], ...updates }
              }
            })),

          setActiveDashboard: (id) =>
            set({ activeDashboard: id }),

          getActivePanels: () => {
            const state = get();
            if (!state.activeDashboard) return [];
            return Object.values(state.panels).filter(
              p => p.dashboardId === state.activeDashboard
            );
          }
        }),
        { limit: 50 } // Time-travel limit
      ),
      { name: 'dashboard-storage' } // Persistence key
    )
  )
);
```

For persistence strategies, see Section 5. For performance optimization, see Section 6.

> **Note**: This section consolidates state management information. Technology recommendations and settings management patterns have been integrated here for completeness.

2.4 Plugin Loader

2.4.1 Purpose

Dynamic loading and management of extensions

2.4.2 Core Features

- Hot module replacement (HMR)
- Lazy loading of plugins
- Plugin dependency management
- Sandboxed execution context
- Plugin lifecycle hooks (init, activate, deactivate, destroy)

2.4.3 Architecture Patterns

Pattern	Description	Pros	Cons	Best For
Dynamic Import	ES modules import()	Native; Code splitting; Simple	Limited metadata; No sandboxing	Modern apps, simple plugins
Module Federation	Webpack 5 feature	Independent deployment; Version isolation; Shared deps	Webpack-specific; Complex setup	Micro-frontends
SystemJS	Universal module loader	Format-agnostic; Runtime loading; Import maps	Extra runtime; Less common	Legacy support needed
iframe Sandboxing	Isolated execution	True isolation; Security; Separate context	Communication overhead; Performance; Complex	Untrusted plugins
Web Workers	Background threads	Non-blocking; Isolated; Parallel	No DOM access; Message passing; Limited	CPU-intensive plugins

2.4.4 Library Comparison

Library	Type	Pros	Cons	Bundle Size	Use Case
single-spa	Micro-frontend framework	Framework-agnostic; Lifecycle; Mature	Complex setup; Learning curve	~5KB	Micro-frontends, large apps
qiankun	Micro-frontend (Alibaba)	Sandboxing; CSS isolation; Full-featured	Complex; Chinese docs	~15KB	Enterprise micro-frontends
import-maps	Native import maps	Native; No build; Simple	Browser support; Limited features	0KB	Modern browsers only
SystemJS	Module loader	Format-agnostic; Import maps; Mature	Extra runtime; Less common	~10KB	Legacy support

Library	Type	Pros	Cons	Bundle Size	Use Case
Custom Loader	DIY	Full control; Tailored; Minimal	Development time; Testing	Varies	Specific requirements

2.4.5 Plugin Lifecycle Patterns

Phase	Purpose	Typical Actions
Load	Fetch plugin code	Download, parse, validate
Initialize	Setup plugin	Register components, create instances
Activate	Start plugin	Mount UI, start services, subscribe to events
Deactivate	Pause plugin	Unmount UI, pause services, keep state
Destroy	Cleanup plugin	Remove listeners, free resources, clear state
Update	Hot reload	Preserve state, swap implementation

2.4.6 BI Dashboard Examples

Platform	Plugin System	Implementation
Observable	Runtime imports	• Dynamic import() for modules; • npm: prefix for packages; • Version pinning; • No formal plugin API
Evidence	Component discovery	• File-based plugin system; • Auto-discovery from directories; • Build-time integration; • Svelte components
Count.co	Canvas plugins	• Visualization plugins; • Data connector plugins; • SQL function extensions; • Custom cell types
tldraw	Shape plugins	• Custom shape definitions; • Tool plugins; • UI override plugins; • Runtime registration
Omni Docs	Plugin system	• Markdown plugins; • Custom renderers; • Build-time and runtime plugins; • Plugin manifest

2.4.7 Recommended Architecture

```
// Plugin loader with lifecycle and sandboxing
interface PluginManifest {
  id: string;
  name: string;
  version: string;
  main: string; // Entry point
  dependencies?: Record<string, string>;
  permissions?: string[];
```

```
  activationEvents?: string[];
}

interface Plugin {
  manifest: PluginManifest;
  activate: (context: PluginContext) => void | Promise<void>;
  deactivate?: () => void | Promise<void>;
}

class PluginLoader {
  private plugins = new Map<string, Plugin>();
  private activated = new Set<string>();

  async load(url: string): Promise<void> {
    // Fetch manifest
    const manifestUrl = `${url}/plugin.json`;
    const manifest: PluginManifest = await fetch(manifestUrl).then(r => r.json());

    // Check dependencies
    if (manifest.dependencies) {
      await this.resolveDependencies(manifest.dependencies);
    }

    // Load plugin code
    const module = await import(/* @vite-ignore */ `${url}/${manifest.main}`);
    const plugin: Plugin = {
      manifest,
      ...module.default
    };

    this.plugins.set(manifest.id, plugin);
  }

  async activate(id: string): Promise<void> {
    const plugin = this.plugins.get(id);
    if (!plugin) throw new Error(`Plugin not found: ${id}`);
    if (this.activated.has(id)) return;

    // Create sandboxed context
    const context = this.createContext(plugin);

    // Activate
    await plugin.activate(context);
    this.activated.add(id);

    console.log(`Plugin activated: ${id}`);
  }

  async deactivate(id: string): Promise<void> {
    const plugin = this.plugins.get(id);
    if (!plugin || !this.activated.has(id)) return;
```

```
    if (plugin.deactivate) {
      await plugin.deactivate();
    }

    this.activated.delete(id);
    console.log(`Plugin deactivated: ${id}`);
  }

  private createContext(plugin: Plugin): PluginContext {
    // Create limited API surface based on permissions
    return {
      registerCommand: (cmd) => commandRegistry.register(cmd),
      registerComponent: (comp) => componentRegistry.register(comp),
      // ... other APIs based on permissions
    };
  }

  private async resolveDependencies(deps: Record<string, string>): Promise<void> {
    // Resolve and load dependencies
    for (const [name, version] of Object.entries(deps)) {
      // Check if already loaded
      // Load from CDN or local
      // Version compatibility check
    }
  }
}
```

For security and sandboxing, see Section 7. For hot module replacement, see Section 6.

Chapter 3

Extension System

The framework provides comprehensive extension capabilities through dual languages and multiple extension points.

3.1 Extension Languages

3.2 Custom DSL

3.2.1 Purpose

Declarative, safe UI composition

3.2.2 Features

- Simple, readable syntax for common patterns
- Type-safe by design
- Limited to safe operations
- Compiles to React components
- Hot-reloadable

3.2.3 Example DSL Syntax (conceptual)

```
dashboard "Sales Overview" {
  layout: grid(2, 2)

  panel chart {
    type: line
    data: query("sales.monthly")
    position: (0, 0)
  }

  panel table {
    data: query("sales.top_products")
    position: (1, 0)
  }
}
```

3.3 JavaScript Extensions

3.3.1 Purpose

Full programmatic control for advanced use cases

3.3.2 Features

- Access to extension API
- React component creation
- Custom data transformations
- Integration with external libraries
- Sandboxed execution

3.4 Extension Points

The framework provides multiple extension points for customizing every aspect of the system.

3.5 UI Components

3.5.1 Extension Capabilities

- Custom chart types and visualizations
- Data widgets (tables, cards, metrics)
- Input controls and forms
- Layout containers and panels
- Themes and styling systems

3.5.2 Component Extension Patterns

Pattern	Description	Pros	Cons	Best For
React Component	Standard React component	Familiar; Full ecosystem; Easy integration	React-only; Bundle size	React-based dashboards
Web Component	Custom elements	Framework-agnostic; Encapsulation; Reusable	Less ecosystem; Complexity	Multi-framework support
Plugin API	Declarative config	Simple; Safe; Validated	Less flexible; Limited features	Simple extensions
Render Function	Function returning JSX/HTML	Flexible; Lightweight; Composable	No lifecycle; Manual cleanup	Simple UI elements

3.5.3 BI Dashboard Examples

Platform	Component System	Extension Method
Observable	Reactive cells	• Custom cells with JavaScript; • Import external libraries; • Inline HTML/SVG; • D3.js visualizations
Evidence	Svelte components	• Custom Svelte components in /components; • Markdown with component tags; • SQL + component binding
Count.co	Canvas components	• Custom visualization cells; • SQL-driven components; • React-based extensions; • Drag-and-drop integration
tldraw	Shape components	• Custom shape definitions; • SVG-based rendering; • Tool components; • React shape API

3.5.4 Recommended Architecture

```typescript
// Component extension API
interface ComponentExtension {
  id: string;
  type: 'chart' | 'widget' | 'control' | 'container';
  component: React.ComponentType<any>;
  schema?: JSONSchema; // Props validation
  defaultProps?: Record<string, any>;
  icon?: string;
  category?: string;
}

// Registration
extensionAPI.registerComponent({
  id: 'custom-heatmap',
  type: 'chart',
  component: CustomHeatmap,
  schema: {
    type: 'object',
    properties: {
      data: { type: 'array' },
      colorScheme: { type: 'string', enum: ['viridis', 'plasma'] }
    }
  },
  icon: 'grid',
  category: 'Advanced Charts'
});
```

For component registry details, see Section 1.1. For React integration, see Section 8.1.

3.6 Commands & Command Palette

3.6.1 Core Concepts

- **Command Registry**: All actions exposed as named commands

- **Fuzzy Search**: Quick command discovery with intelligent matching
- **Command History**: Recently used commands for quick access
- **Parameterized Commands**: Commands that accept arguments
- **Keyboard-First**: Fully navigable via keyboard

3.6.2 Architecture Overview

A command palette is a **searchable command interface** that provides: 1. Unified access point for all application actions 2. Fuzzy search for command discovery 3. Keyboard-driven navigation (no mouse required) 4. Context-aware command filtering 5. Command execution with optional parameters

3.6.3 Key Design Decisions

Aspect	Recommended Approach	Rationale
Search Algorithm	Fuzzy matching with ranking	Handles typos, partial matches, and prioritizes relevance
Command Structure	Hierarchical with categories	Organizes commands logically, supports grouping
Activation	Global hotkey (Cmd/Ctrl+K)	Industry standard, muscle memory from other tools
UI Pattern	Modal overlay with input + list	Focuses attention, doesn't disrupt workflow
Performance	Virtual scrolling for large lists	Handles 1000+ commands without lag
Extensibility	Plugin-contributed commands	Extensions can register custom commands

3.6.4 Architecture Patterns

1. **Command Registration Pattern**

```
interface Command {
  id: string;
  name: string;
  description?: string;
  category?: string;
  keywords?: string[];
  icon?: string;
  execute: (args?: any) => void | Promise<void>;
  when?: () => boolean; // Context condition
}
```

2. **Search Ranking Strategy**

   - Exact match (highest priority)
   - Prefix match
   - Fuzzy match with position weighting
   - Keyword match
   - Recent usage boost
   - Frequency boost

3. **UI State Management**
    - Open/closed state
    - Search query
    - Selected command index
    - Filtered & ranked results
    - Command history

3.6.5 Library Comparison

Library	Type	Pros	Cons	Bundle Size	Use Case
kbar	React component	Beautiful UI; Nested actions; TypeScript; Animations; Active development	React-only; Opinionated styling; Limited customization	~15KB	Modern React apps, design-focused
cmdk	React primitive	Headless/unstyled; Flexible; Accessible; Small bundle; By Vercel	React-only; Requires styling; More setup needed	~8KB	Custom designs, full control
ninja-keys	Web component	Framework-agnostic; Web components; Zero dependencies; Easy integration	Less flexible; Styling limitations; Smaller ecosystem	~12KB	Multi-framework, simple needs
command-score	Algorithm only	Just scoring logic; Framework-agnostic; Tiny size; Fast	No UI; Build everything yourself; More work	~2KB	Custom implementations
Fuse.js	Fuzzy search	Powerful search; Configurable; Framework-agnostic; Mature	No UI; Larger bundle; Overkill for simple cases	~20KB	Complex search requirements

Library	Type	Pros	Cons	Bundle Size	Use Case
Custom Build	DIY	Full control; Minimal bundle; Tailored features; No dependencies	Development time; Maintenance burden; Reinventing wheel	Varies	Unique requirements, learning

3.6.6 Fuzzy Search Algorithm Comparison

Algorithm	Approach	Pros	Cons	Best For
Substring Match	Simple `includes()`	Fast; Simple; Predictable	No typo tolerance; No ranking; Order-dependent	Simple lists, exact matching
Levenshtein Distance	Edit distance	Typo-tolerant; Well-understood; Good ranking	Slower ($O(n^2)$); No position weighting	Spell-check, small datasets
Fuzzy Matching (fzy, fzf-style)	Character sequence	Fast; Position-aware; Intuitive results; Handles abbreviations	More complex; Tuning needed	Command palettes, file search
N-gram Based	Token matching	Language-aware; Good for text; Handles word order	Slower; More memory; Complex setup	Full-text search, documents

3.6.7 BI Dashboard Examples

Platform	Implementation	Features	Activation
Observable	Custom React component	• Fuzzy search across notebooks; • Recent notebooks; • Command suggestions; • Cell navigation; • Keyboard shortcuts reference	Cmd/Ctrl+K
Evidence	Custom implementation	• Page navigation; • Component search; • Query execution; • Documentation search; • Settings access	Cmd/Ctrl+K

Platform	Implementation	Features	Activation
Count.co	Custom React	• Canvas search; • Cell navigation; • Query execution; • Data source selection; • Quick actions	Cmd/Ctrl+K
tldraw	Custom implementation	• Shape search; • Tool selection; • Canvas actions; • Quick commands; • Keyboard shortcuts	Cmd/Ctrl+K
Linear (inspiration)	cmdk-based	• Issue search; • Project navigation; • Quick actions; • Nested commands; • Beautiful animations	Cmd/Ctrl+K

3.6.8 Recommended Architecture for BI Dashboards

```
// Command palette architecture
interface CommandPaletteState {
  isOpen: boolean;
  query: string;
  selectedIndex: number;
  commands: Command[];
  filteredCommands: Command[];
  recentCommands: string[];
  mode: 'commands' | 'dashboards' | 'search';
}

// Multi-mode support (like VS Code)
const modes = {
  commands: {
    prefix: '>',
    placeholder: 'Type a command...',
    source: () => getAllCommands()
  },
  dashboards: {
    prefix: '',
    placeholder: 'Search dashboards...',
    source: () => getDashboards()
  },
  search: {
    prefix: '/',
    placeholder: 'Search data...',
    source: (query) => searchData(query)
  }
};
```

3.6.9 Performance Optimization Strategies

Strategy	Technique	Impact
Virtual Scrolling	Render only visible items	Handles 10,000+ commands smoothly
Debounced Search	Delay search by 150-300ms	Reduces unnecessary computations
Memoized Results	Cache search results	Faster re-renders on navigation
Web Workers	Offload search to worker thread	Keeps UI responsive for large datasets
Indexed Commands	Pre-build search index	Sub-millisecond lookups
Lazy Loading	Load command metadata on-demand	Faster initial load

3.6.10 Accessibility Considerations

- **ARIA Labels**: Proper roles and labels for screen readers
- **Keyboard Navigation**: Arrow keys, Enter, Escape, Tab
- **Focus Management**: Trap focus in modal, restore on close
- **Announcements**: Screen reader feedback for results count
- **High Contrast**: Support for high contrast themes
- **Reduced Motion**: Respect `prefers-reduced-motion`

3.6.11 Advanced Features

1. **Nested Commands** (Breadcrumb navigation)
   - Parent command opens sub-menu
   - Back navigation with Escape
   - Visual breadcrumb trail
2. **Command Scoring**
   - Frecency (frequency + recency)
   - User preference learning
   - Context-aware boosting
3. **Multi-Step Commands**
   - Command with parameter prompts
   - Wizard-like flows
   - Validation and error handling
4. **Command Chaining**
   - Execute multiple commands in sequence
   - Macro recording
   - Batch operations

3.6.12 Recommended Stack for Your Framework

- **React Apps**: `cmdk` (headless) or `kbar` (styled)
- **Framework-Agnostic**: `ninja-keys` or custom build
- **Search Algorithm**: Fuzzy matching (fzy-style) with position weighting
- **UI Pattern**: Modal overlay with virtual scrolling

- **State Management**: Zustand or local React state
- **Persistence**: Recent commands in LocalStorage, frecency in IndexedDB
- **Activation**: Cmd/Ctrl+K (global), with mode switching support

3.6.13 Implementation Checklist

☐ Command registration system
☐ Fuzzy search with ranking
☐ Keyboard navigation (↑↓ Enter Esc)
☐ Virtual scrolling for performance
☐ Recent commands tracking
☐ Context-aware filtering
☐ Multi-mode support (commands/search/navigation)
☐ Accessibility (ARIA, focus management)
☐ Visual feedback (loading, no results)
☐ Mobile support (optional for BI dashboards)

For integration with keybindings, see Section 2.2.3. For state management patterns, see Section 9.1.1.

3.6.14 Extension Capabilities

- Custom actions and workflows
- Data processing pipelines
- External API integrations
- Batch operations
- Scheduled tasks

3.6.15 Command Extension Patterns

Pattern	Description	Use Case
Simple Command	Single action	Quick operations
Parameterized	Accepts arguments	Flexible actions
Async Command	Promise-based	API calls, long operations
Composite	Multiple sub-commands	Complex workflows
Scheduled	Cron-like execution	Periodic tasks

3.6.16 BI Dashboard Examples

Platform	Command System	Extension Method
Observable	Cell execution	• Cells as commands; • Function exports; • Import and call
Evidence	Build commands	• npm scripts; • CLI commands; • No runtime commands
Count.co	Canvas commands	• Cell execution commands; • Query commands; • Canvas actions
tldraw	Tool commands	• Shape commands; • Canvas commands; • Tool actions

Platform	Command System	Extension Method
VS Code	Commands API	• commands.registerCommand; • Command palette; • Keybinding integration

3.6.17 Recommended API

```
// Command registration
extensionAPI.registerCommand({
  id: 'export-to-pdf',
  name: 'Export Dashboard to PDF',
  category: 'Export',
  execute: async (context) => {
    const dashboard = context.getActiveDashboard();
    const pdf = await generatePDF(dashboard);
    await downloadFile(pdf, 'dashboard.pdf');
  },
  when: (context) => context.hasActiveDashboard(),
  icon: 'download'
});

// Pipeline command
extensionAPI.registerPipeline({
  id: 'data-transform',
  steps: [
    { command: 'fetch-data', params: { source: 'api' } },
    { command: 'transform-data', params: { type: 'aggregate' } },
    { command: 'update-panel', params: { panelId: 'main' } }
  ]
});
```

For command palette integration, see Section 2.2.2.

3.7 Keybindings

3.7.1 Core Concepts

- **Global Keybindings**: System-wide keyboard shortcuts (always active)
- **Mode-Specific Keybindings**: Context-aware shortcuts based on active component
- **Keymaps**: Hierarchical keybinding definitions with priority resolution
- **Chord Support**: Multi-key sequences (e.g., `Ctrl+x Ctrl+s`)
- **Customizable**: Users can rebind any key combination

3.7.2 Architecture Overview

The keybinding system follows a **command-based architecture** where: 1. Commands are first-class entities with unique IDs 2. Keybindings map to commands (many-to-one relationship) 3. Context evaluation determines which bindings are active 4. Priority hierarchy resolves conflicts (Local → Mode → Global)

3.7.3 Key Design Decisions

Aspect	Recommended Approach	Rationale
Command Registry	Centralized registry with command objects	Enables command palette, customization, and discoverability
Context Awareness	"When" clauses for conditional activation	Allows same key to trigger different commands based on context
Conflict Resolution	Priority-based hierarchy with context evaluation	Predictable behavior while supporting complex scenarios
Chord Sequences	Support multi-key sequences with timeout	Expands available key combinations for power users
Persistence	Store custom bindings in IndexedDB	User customizations persist across sessions

3.7.4 Library Recommendations by Use Case

- **Minimal Bundle Size** (<1KB): `tinykeys` - 400 bytes, chord support, zero dependencies
- **Feature-Rich** (~3KB): `hotkeys-js` - scope support, filtering, mature ecosystem
- **React Integration** (~2KB): `react-hotkeys-hook` - hooks-based, component-scoped
- **Full Control**: Custom implementation - tailored to exact needs, no dependencies

3.7.5 BI Dashboard Examples

- **Observable**: Command palette (Cmd+K) with fuzzy search, cell-specific shortcuts, notebook-level keybindings
- **Evidence**: Vim-like modal keybindings for power users, context-aware navigation
- **Count.co**: Canvas shortcuts (navigation, cell execution, query editing), SQL editor keybindings
- **tldraw**: Canvas shortcuts (shape creation, selection, transformation), tool-specific bindings
- **VS Code** (pattern): Comprehensive "when" clause system, keybinding editor UI

See Section 9.1.2 for detailed architectural analysis, pros/cons comparison, and implementation strategies.

3.7.6 Extension Capabilities

- Custom keyboard shortcuts
- Mode-specific bindings
- Chord sequences (multi-key)
- Macro recording and playback
- Context-aware activation

3.7.7 Keybinding Extension Patterns

Approach	Implementation	Pros	Cons
Declarative	JSON/YAML config	Simple; Validated; Safe	Limited logic; No dynamic binding
Programmatic	API calls	Flexible; Dynamic; Conditional	More complex; Error-prone

Approach	Implementation	Pros	Cons
Hybrid	Config + API	Best of both; Validated + flexible	Two systems

3.7.8 BI Dashboard Examples

Platform	Keybinding System	Extension Method
Observable	Built-in shortcuts	• Limited customization; • Cell-level shortcuts; • Cmd+K command palette
Evidence	Not extensible	• Fixed keybindings; • No custom shortcuts
Count.co	Canvas shortcuts	• Cell navigation keys; • Query execution shortcuts; • Custom keybindings
tldraw	Tool shortcuts	• Shape creation keys; • Canvas navigation; • Tool-specific bindings
VS Code	Keybindings API	• keybindings.json; • when clauses; • Full customization

3.7.9 Recommended API

```
// Keybinding extension
extensionAPI.registerKeybinding({
  key: 'Ctrl+Shift+E',
  command: 'export-dashboard',
  when: 'dashboardActive && !editing',
  description: 'Export current dashboard'
});

// Macro support
extensionAPI.registerMacro({
  name: 'refresh-all-panels',
  keys: ['Ctrl+R', 'Ctrl+A'],
  actions: [
    { command: 'select-all-panels' },
    { command: 'refresh-panels' }
  ]
});
```

For keybinding architecture, see Section 2.2.3 and 9.1.2.

Note: Detailed keybinding architecture consolidated here from multiple sections.

3.8 Themes

3.8.1 Extension Capabilities

- Color schemes and palettes

- Typography systems
- Component styling
- Dark/light mode variants
- Custom CSS variables

3.8.2 Theme Extension Patterns

Approach	Implementation	Pros	Cons
CSS Variables	Override root variables	Simple; Performant; Dynamic	Limited scope; No logic
Theme Object	JavaScript config	Type-safe; Validated; Programmatic	Runtime overhead; More complex
CSS File	Separate stylesheet	Familiar; Standard; Cacheable	No dynamic; Load overhead
Hybrid	Variables + Object	Flexible; Best of both	Complexity

3.8.3 BI Dashboard Examples

Platform	Theme System	Extension Method
Observable	CSS variables	• Custom CSS in cells; • Theme cells; • CSS imports
Evidence	Tailwind config	• tailwind.config.js; • Custom CSS; • Component styling
Count.co	Theme settings	• Color customization; • Canvas themes; • CSS variables
tldraw	Theme system	• CSS variables; • Custom themes; • Dark/light mode; • Color overrides

3.8.4 Recommended API

```
// Theme registration
extensionAPI.registerTheme({
  id: 'ocean-blue',
  name: 'Ocean Blue',
  type: 'dark',
  colors: {
    primary: '#0077be',
    background: '#001f3f',
    text: '#ffffff',
    // ... more colors
  },
  typography: {
```

```
    fontFamily: 'Inter, sans-serif',
    fontSize: {
      base: '14px',
      heading: '24px'
    }
  },
  shadows: {
    sm: '0 1px 2px rgba(0,0,0,0.1)',
    md: '0 4px 6px rgba(0,0,0,0.1)'
  }
});
```

For theme architecture, see Section 9.1.3.

3.9 Layouts

3.9.1 Buffer/Window Model

- **Buffers**: Logical content containers (data views, charts, tables)
- **Windows**: Visual panes that display buffers
- **Layout Management**: Split, resize, and arrange windows dynamically
- **Buffer Switching**: Quick navigation between different data views

3.9.2 Extension Capabilities

- Window arrangements
- Dashboard templates
- Responsive breakpoints
- Grid configurations
- Split pane layouts

3.9.3 Layout Extension Patterns

Pattern	Description	Use Case
Template	Predefined layout	Quick start dashboards
Programmatic	Code-defined layout	Dynamic layouts
Declarative	JSON/YAML config	Shareable layouts
Interactive	Drag-and-drop	User customization

3.9.4 BI Dashboard Examples

Platform	Layout System	Extension Method
Observable	Notebook flow	• Linear cell layout; • Custom layouts via HTML; • Grid layouts in cells
Evidence	Page templates	• Markdown-based; • Component slots; • Fixed layouts

Platform	Layout System	Extension Method
Count.co	Canvas layout	• Free-form canvas; • Cell positioning; • Auto-layout options; • Responsive grids
tldraw	Canvas system	• Infinite canvas; • Shape positioning; • Grouping and frames; • Custom layouts

3.9.5 Recommended API

```
// Layout template registration
extensionAPI.registerLayoutTemplate({
  id: 'analytics-dashboard',
  name: 'Analytics Dashboard',
  description: '3-column layout with header',
  thumbnail: '/templates/analytics.png',
  layout: {
    type: 'grid',
    cols: 12,
    rows: 'auto',
    items: [
      { id: 'header', x: 0, y: 0, w: 12, h: 2, component: 'header' },
      { id: 'sidebar', x: 0, y: 2, w: 3, h: 10, component: 'sidebar' },
      { id: 'main', x: 3, y: 2, w: 9, h: 10, component: 'main' }
    ]
  },
  responsive: {
    mobile: { cols: 1 },
    tablet: { cols: 6 },
    desktop: { cols: 12 }
  }
});
```

For layout architecture, see Section 9.1.4.

3.10 Hooks & Advice

3.10.1 Core Concepts

Hooks are extension points that allow plugins to inject custom behavior at specific lifecycle events without modifying core code.

Advice is a technique to wrap or modify existing functions, enabling plugins to intercept, augment, or replace behavior.

3.10.2 Architecture Overview

The hooks and advice system provides a **plugin architecture** that enables: 1. Decoupled extension points throughout the application 2. Multiple handlers for the same event (observer pattern) 3. Function interception and modification (aspect-oriented programming) 4. Plugin lifecycle management 5. Predictable execution order

3.10.3 Key Design Decisions

Aspect	Recommended Approach	Rationale
Hook Registration	Event emitter pattern with typed events	Type-safe, familiar pattern, supports multiple listeners
Execution Order	Priority-based with explicit ordering	Predictable behavior, handles dependencies
Error Handling	Isolated execution with error boundaries	One plugin failure doesn't break others
Async Support	Promise-based hooks with timeout	Handles async operations, prevents hanging
Advice Pattern	Middleware/decorator pattern	Composable, chainable, familiar to developers
Unsubscribe	Return cleanup function	Prevents memory leaks, follows React patterns

3.10.4 Hook Types & Use Cases

Hook Category	Examples	Use Cases
Lifecycle Hooks	`app-init`, `app-ready`, `app-destroy`	Initialize services, cleanup resources
Render Hooks	`before-render`, `after-render`, `render-error`	Inject UI, track performance, error handling
State Hooks	`before-state-change`, `after-state-change`, `state-hydrate`	Validation, logging, persistence
Plugin Hooks	`plugin-loaded`, `plugin-activated`, `plugin-unloaded`	Plugin coordination, dependency management
Data Hooks	`before-query`, `after-query`, `query-error`	Data transformation, caching, error handling
Navigation Hooks	`before-navigate`, `after-navigate`, `route-change`	Analytics, guards, breadcrumbs
User Action Hooks	`command-execute`, `keybinding-trigger`, `menu-click`	Analytics, macros, automation

3.10.5 Architecture Patterns

1. **Event Emitter Pattern** (Hooks)

```typescript
interface HookSystem {
  on(event: string, handler: Function, priority?: number): () => void;
  emit(event: string, ...args: any[]): Promise<void>;
  once(event: string, handler: Function): () => void;
  off(event: string, handler: Function): void;
```

```
}

// Usage
const unsubscribe = hooks.on('before-render', (context) => {
  console.log('Rendering:', context.component);
});
```

2. **Middleware Pattern** (Advice)

```
type Middleware<T> = (
  context: T,
  next: () => Promise<any>
) => Promise<any>;

// Usage
const loggingMiddleware: Middleware<QueryContext> = async (ctx, next) => {
  console.log('Query start:', ctx.query);
  const result = await next();
  console.log('Query end:', result);
  return result;
};
```

3. **Decorator Pattern** (Advice)

```
function withLogging(fn: Function) {
  return function(...args: any[]) {
    console.log('Before:', args);
    const result = fn.apply(this, args);
    console.log('After:', result);
    return result;
  };
}
```

3.10.6 Library Comparison

Library	Type	Pros	Cons	Bundle Size	Use Case
mitt	Event emitter	Tiny (200B); Simple API; TypeScript support; No dependencies	No priority ordering; No async handling; Basic features only	200B	Lightweight hooks, simple events
eventemitter3	Event emitter	Fast performance; Mature; Node.js compatible; Well-tested	Larger bundle; No TypeScript out of box; No priority support	~2KB	Production apps, performance-critical

Library	Type	Pros	Cons	Bundle Size	Use Case
nanoevents	Event emitter	Very small (200B); Simple; TypeScript support	Minimal features; No wildcards; No priority	200B	Minimal footprint, basic needs
hookified	Hook system	Built for plugins; Priority support; Async hooks; Typed	Less popular; Smaller ecosystem	~3KB	Plugin systems, complex hooks
Tapable	Hook system	Webpack's hook system; Very powerful; Multiple hook types; Battle-tested	Complex API; Large bundle; Steep learning curve	~10KB	Complex plugin systems, Webpack-like
Custom Build	DIY	Tailored features; Minimal size; Full control	Development time; Testing needed; Maintenance	Varies	Specific requirements

3.10.7 Hook System Implementation Patterns

Pattern	Description	Pros	Cons
Simple Event Emitter	Basic pub/sub	Simple; Familiar; Small	No ordering; No async control
Priority Queue	Ordered execution by priority	Predictable order; Dependency handling	More complex; Priority conflicts
Async Waterfall	Sequential async execution	Data transformation; Pipeline pattern	Slower; Error propagation
Async Parallel	Concurrent execution	Fast; Independent handlers	No ordering; Race conditions
Async Series	Sequential with results	Ordered; Result aggregation	Slower; Blocking

3.10.8 Advice Pattern Comparison

Pattern	Implementation	Pros	Cons	Best For
Proxy-Based	ES6 Proxy	Transparent; No code changes; Powerful	Performance overhead; Debugging harder; Browser support	Dynamic interception
Decorator-Based	Function wrapping	Explicit; Composable; TypeScript support	Requires wrapping; Boilerplate	Explicit augmentation
Middleware Chain	Express-style	Familiar pattern; Composable; Async-friendly	More setup; Context passing	Request/response flows
AOP Framework	AspectJ-style	Powerful; Declarative; Cross-cutting	Complex; Large bundle; Learning curve	Enterprise apps

3.10.9 BI Dashboard Examples

Platform	Hook System	Advice System	Implementation Details
Observable	Custom event system	Runtime notebook hooks	• Cell execution hooks; • Import hooks; • Reactive dependency tracking; • Custom hook for data loading
Evidence	Component lifecycle	Build-time hooks	• Page build hooks; • Component mount/unmount; • Data query hooks; • Markdown processing hooks
Count.co	Canvas lifecycle	Canvas hooks	• Cell execution hooks; • Query lifecycle hooks; • Canvas render hooks; • Collaboration hooks
tldraw	Shape lifecycle	Shape hooks	• Shape creation/update hooks; • Canvas interaction hooks; • History hooks (undo/redo); • Selection hooks
VS Code	Extension API	Command/menu contribution	• Activation events; • Language server hooks; • Workspace events; • Decoration providers
Webpack	Tapable hooks	Compilation hooks	• Compiler hooks; • Compilation hooks; • Module hooks; • Asset optimization

3.10.10 Recommended Architecture for BI Dashboards

```
// Hook system with priority and async support
class HookSystem {
```

```
  private hooks = new Map<string, Hook[]>();

  on(event: string, handler: Function, priority = 10): () => void {
    if (!this.hooks.has(event)) {
      this.hooks.set(event, []);
    }

    const hook = { handler, priority };
    const hooks = this.hooks.get(event)!;

    // Insert by priority (higher = earlier)
    const index = hooks.findIndex(h => h.priority < priority);
    if (index === -1) {
      hooks.push(hook);
    } else {
      hooks.splice(index, 0, hook);
    }

    // Return unsubscribe function
    return () => {
      const hooks = this.hooks.get(event);
      if (hooks) {
        const idx = hooks.indexOf(hook);
        if (idx > -1) hooks.splice(idx, 1);
      }
    };
  }

  async emit(event: string, context: any): Promise<any> {
    const hooks = this.hooks.get(event) || [];
    let result = context;

    for (const { handler } of hooks) {
      try {
        const handlerResult = await handler(result);
        // Allow handlers to transform data
        if (handlerResult !== undefined) {
          result = handlerResult;
        }
      } catch (error) {
        console.error(`Hook error in ${event}:`, error);
        // Continue with other hooks
      }
    }

    return result;
  }
}

// Advice system with middleware pattern
class AdviceSystem {
```

```
  private advices = new Map<string, Middleware[]>();

  addAdvice(target: string, middleware: Middleware): () => void {
    if (!this.advices.has(target)) {
      this.advices.set(target, []);
    }

    this.advices.get(target)!.push(middleware);

    return () => {
      const advices = this.advices.get(target);
      if (advices) {
        const idx = advices.indexOf(middleware);
        if (idx > -1) advices.splice(idx, 1);
      }
    };
  }

  async execute(target: string, context: any, original: Function): Promise<any> {
    const advices = this.advices.get(target) || [];

    // Build middleware chain
    let index = 0;
    const next = async (): Promise<any> => {
      if (index < advices.length) {
        const middleware = advices[index++];
        return middleware(context, next);
      } else {
        // Execute original function
        return original(context);
      }
    };

    return next();
  }
}
```

3.10.11 Common Hook Patterns

1. **Data Transformation Pipeline**

```
// Transform data through multiple plugins
hooks.on('data-transform', (data) => {
  return { ...data, transformed: true };
});

const result = await hooks.emit('data-transform', rawData);
```

2. **Validation Chain**

```
// Validate with multiple validators
hooks.on('validate-dashboard', (dashboard) => {
```

```
  if (!dashboard.title) throw new Error('Title required');
  return dashboard;
});
```

3. **Lifecycle Management**

```
// Plugin initialization
hooks.on('plugin-loaded', async (plugin) => {
  await plugin.initialize();
  console.log(`Plugin ${plugin.name} loaded`);
});
```

4. **Conditional Execution**

```
// Execute only if condition met
hooks.on('before-save', (data) => {
  if (data.needsValidation) {
    return validate(data);
  }
  return data;
});
```

3.10.12 Performance Considerations

Concern	Strategy	Impact
Too Many Hooks	Debounce/throttle high-frequency hooks	Reduces CPU usage
Slow Handlers	Timeout enforcement	Prevents hanging
Memory Leaks	Automatic cleanup on plugin unload	Prevents memory growth
Error Propagation	Isolated execution with try/catch	Stability
Async Coordination	Promise.all for parallel, sequential for order	Performance vs order

3.10.13 Security Considerations

- **Sandboxing**: Execute plugin hooks in isolated context
- **Capability Checks**: Verify plugin has permission for hook
- **Input Validation**: Sanitize data passed to hooks
- **Timeout Enforcement**: Prevent infinite loops
- **Resource Limits**: Cap memory/CPU usage per hook

3.10.14 Advanced Features

1. **Hook Composition**
   - Combine multiple hooks into one
   - Reusable hook patterns
   - Hook inheritance
2. **Conditional Hooks**
   - Execute only when condition met
   - Context-aware activation
   - Dynamic hook registration

3. **Hook Debugging**
    - Hook execution tracing
    - Performance profiling
    - Dependency visualization
4. **Hook Versioning**
    - API version compatibility
    - Deprecation warnings
    - Migration helpers

3.10.15 Recommended Stack

- **Simple Hooks**: `mitt` (200B) or `eventemitter3` (~2KB)
- **Complex Plugin System**: `Tapable` or custom implementation
- **Advice/Middleware**: Custom middleware chain
- **Type Safety**: TypeScript with strict typing
- **Error Handling**: Isolated execution with error boundaries
- **Async**: Promise-based with timeout enforcement

3.10.16 Implementation Checklist

☐ Hook registration system with priority
☐ Event emitter with typed events
☐ Async hook support with timeout
☐ Error isolation per handler
☐ Unsubscribe/cleanup mechanism
☐ Middleware/advice pattern
☐ Plugin lifecycle hooks
☐ Documentation for hook points
☐ Debugging/tracing tools
☐ Performance monitoring

For plugin architecture, see Section 1.4. For extension patterns, see Section 2.1.

3.11 Hot Reloading

3.11.1 Overview

Hot Module Replacement (HMR) enables developers to update extensions in real-time without losing application state, dramatically improving development velocity.

3.11.2 Core Capabilities

- Live code updates without page refresh
- State preservation across reloads
- Error recovery and isolation
- Enhanced debugging and error reporting

3.11.3 HMR Architecture Patterns

Pattern	Description	Pros	Cons	Best For
Full Reload	Refresh entire page	Simple; Reliable; No state issues	Slow; Loses state; Poor DX	Production, simple apps
Module HMR	Replace individual modules	Fast; Preserves state; Good DX	Complex; State management; Edge cases	Development, modern apps
Component HMR	Replace React components	Very fast; Preserves local state; Best DX	React-specific; Requires setup	React development
Live Reload	Watch files, auto-refresh	Simple; Universal; Reliable	Loses state; Slower; Full reload	Simple development

3.11.4 HMR Library Comparison

Tool	Type	Pros	Cons	Use Case
Vite HMR	Build tool	Very fast; ESM-based; Simple API; React Fast Refresh	Vite-specific; Modern browsers only	Modern React/Vue apps
Webpack HMR	Build tool	Mature; Powerful; Ecosystem; Configurable	Complex; Slower; Large config	Enterprise apps
Parcel HMR	Build tool	Zero config; Fast; Automatic	Less control; Smaller ecosystem	Quick prototypes
React Fast Refresh	React HMR	Preserves state; Error recovery; Best DX	React-only; Requires setup	React development
Custom HMR	DIY	Full control; Tailored	Complex; Maintenance	Unique requirements

3.11.5 State Preservation Strategies

Strategy	Implementation	Pros	Cons
Local State	Component state preserved	Automatic; Simple	Only local state; Limited
Global State	Store persisted	Full state; Reliable	Manual setup; Serialization
Snapshot	Save/restore state	Complete; Flexible	Complex; Performance

Strategy	Implementation	Pros	Cons
Hybrid	Critical state persisted	Balanced; Optimized	More logic

3.11.6 BI Dashboard Examples

Platform	HMR System	Implementation
Observable	Live evaluation	• Cell re-execution on change; • Reactive dependency tracking; • Instant feedback; • State in cells
Evidence	Vite HMR	• Vite dev server; • Svelte HMR; • Fast refresh; • Component state preserved
Count.co	Vite HMR	• React Fast Refresh; • Canvas state preservation; • Cell hot reload; • Query result caching
tldraw	Vite HMR	• Shape state preservation; • Canvas hot reload; • Tool hot swap; • History preservation

3.11.7 Error Recovery Patterns

Pattern	Description	Use Case
Error Boundary	React error boundaries	Isolate component errors
Fallback UI	Show error state	User-friendly error display
Auto-Retry	Retry failed reload	Transient errors
Rollback	Revert to last working	Critical failures

3.11.8 Recommended Architecture

```
// HMR integration for extensions
if (import.meta.hot) {
  import.meta.hot.accept((newModule) => {
    // Preserve state
    const currentState = extensionAPI.getState();

    // Unload old extension
    extensionAPI.unload(extensionId);

    // Load new extension
    extensionAPI.load(newModule.default);

    // Restore state
    extensionAPI.setState(currentState);
```

```
    console.log('Extension hot reloaded:', extensionId);
  });

  import.meta.hot.dispose(() => {
    // Cleanup before reload
    extensionAPI.cleanup(extensionId);
  });
}

// Error recovery
window.addEventListener('error', (event) => {
  if (event.filename?.includes('/extensions/')) {
    // Extension error - isolate and recover
    extensionAPI.handleError(event.error);
    event.preventDefault();
  }
});
```

3.11.9 Development Mode Features

- **Source Maps**: Map compiled code to source for debugging
- **Error Overlay**: Full-screen error display with stack traces
- **Console Integration**: Enhanced logging with extension context
- **Performance Profiling**: Track reload times and bottlenecks
- **State Inspector**: Visualize state changes
- **Network Monitoring**: Track extension API calls

For build tool configuration, see Section 6. For plugin lifecycle, see Section 1.4.

3.12 Live Evaluation of DSL & JavaScript

3.12.1 Overview

The framework supports live evaluation and hot reloading of both DSL and JavaScript code blocks within dashboards, enabling rapid development and interactive data exploration.

3.12.2 Core Capabilities

- Real-time DSL compilation and component rendering
- Sandboxed JavaScript execution with reactive dependencies
- State preservation across code changes
- Cell-based evaluation with dependency tracking
- Incremental compilation for performance

3.12.3 JavaScript Live Evaluation

3.12.3.1 Architecture Pattern

Observable-style reactive cells with sandboxed execution

3.12.3.2 Implementation

```
// Cell-based JavaScript evaluator
interface Cell {
```

```
  id: string;
  type: 'dsl' | 'javascript';
  code: string;
  dependencies: string[];
  output?: any;
  error?: Error;
}

class LiveJSEvaluator {
  private cells = new Map<string, Cell>();
  private graph = new DependencyGraph();

  // Evaluate JavaScript in sandboxed context
  async evaluateCell(cellId: string, context: Record<string, any>) {
    const cell = this.cells.get(cellId);
    if (!cell) throw new Error(`Cell not found: ${cellId}`);

    try {
      // Create sandboxed function
      const fn = new Function(...Object.keys(context), cell.code);
      cell.output = await fn(...Object.values(context));
      cell.error = undefined;

      // Re-evaluate dependent cells
      const dependents = this.graph.getDependents(cellId);
      for (const depId of dependents) {
        await this.evaluateCell(depId, context);
      }
    } catch (error) {
      cell.error = error as Error;
      throw error;
    }
  }

  // Watch for code changes and auto-evaluate
  watch(cellId: string, context: Record<string, any>) {
    this.on(`cell:${cellId}:change`, () => {
      this.evaluateCell(cellId, context);
    });
  }

  // HMR integration
  enableHMR(cellId: string) {
    if (import.meta.hot) {
      import.meta.hot.accept(() => {
        const cell = this.cells.get(cellId);
        if (cell) {
          this.evaluateCell(cellId, this.getContext());
        }
      });
    }
```

```
  }
}
```

3.12.3.3 Security Integration

```
// Safe evaluation with permission checks
class SecureJSEvaluator extends LiveJSEvaluator {
  constructor(private permissionManager: PermissionManager) {
    super();
  }

  async evaluateCell(cellId: string, context: Record<string, any>) {
    const cell = this.cells.get(cellId);

    // Check permissions
    if (!this.permissionManager.check(cellId, 'system:eval')) {
      throw new Error('Permission denied: system:eval required for JavaScript execution');
    }

    // Create restricted context
    const sandbox = this.createSandbox(cellId);
    const restrictedContext = this.filterContext(context, cellId);

    return super.evaluateCell(cellId, restrictedContext);
  }

  private createSandbox(cellId: string): SandboxContext {
    const permissions = this.permissionManager.getPermissions(cellId);
    return createSandboxedAPI(cellId, permissions);
  }
}
```

3.12.4 DSL Live Compilation

3.12.4.1 Architecture Pattern

Compile-to-React with Vite HMR integration

3.12.4.2 Implementation

```
// DSL live compiler with hot reload
class DSLLiveCompiler {
  private cache = new Map<string, CompiledComponent>();
  private parser = new DSLParser();
  private compiler = new DSLToReactCompiler();

  // Compile DSL to React component
  compile(dslCode: string, moduleId: string): React.ComponentType {
    // Check cache
    if (this.cache.has(moduleId)) {
      return this.cache.get(moduleId)!.component;
    }
```

```
  // 1. Parse DSL
  const ast = this.parser.parse(dslCode);

  // 2. Compile to React
  const component = this.compiler.toReact(ast);

  // 3. Cache result
  this.cache.set(moduleId, { ast, component, code: dslCode });

  // 4. Enable HMR
  this.enableHMR(moduleId, dslCode);

  return component;
}

// Incremental compilation for performance
recompile(moduleId: string, newCode: string): React.ComponentType {
  const cached = this.cache.get(moduleId);

  if (cached) {
    // Parse new code
    const newAST = this.parser.parse(newCode);

    // Compute diff
    const diff = this.differ.diff(cached.ast, newAST);

    // Only recompile changed parts
    if (diff.isMinimal) {
      const component = this.compiler.partialCompile(cached.component, diff);
      this.cache.set(moduleId, { ast: newAST, component, code: newCode });
      return component;
    }
  }

  // Full recompile
  return this.compile(newCode, moduleId);
}

// HMR integration
private enableHMR(moduleId: string, dslCode: string) {
  if (import.meta.hot) {
    import.meta.hot.accept(moduleId, (newModule) => {
      // Preserve component state
      const state = this.getComponentState(moduleId);

      // Recompile
      const newComponent = this.recompile(moduleId, newModule.code);

      // Restore state
      this.setComponentState(moduleId, state);
```

```
        console.log(`DSL hot reloaded: ${moduleId}`);
      });
    }
  }

  // Watch file system for changes
  watch(dslFile: string) {
    const watcher = fs.watch(dslFile, (eventType) => {
      if (eventType === 'change') {
        const code = fs.readFileSync(dslFile, 'utf-8');
        this.recompile(dslFile, code);
      }
    });

    return () => watcher.close();
  }
}
```

3.12.5 Hybrid Cell-Based System

3.12.5.1 Recommended Architecture

Combines DSL and JavaScript evaluation with unified state management

3.12.5.2 Implementation

```
// Unified live dashboard engine
class LiveDashboardEngine {
  private dslCompiler = new DSLLiveCompiler();
  private jsEvaluator = new SecureJSEvaluator(permissionManager);
  private stateManager = useDashboardStore();

  // Evaluate any cell type
  async evaluateCell(cell: Cell) {
    if (cell.type === 'dsl') {
      return this.evaluateDSL(cell);
    } else if (cell.type === 'javascript') {
      return this.evaluateJS(cell);
    }
    throw new Error(`Unknown cell type: ${cell.type}`);
  }

  // DSL evaluation
  private async evaluateDSL(cell: Cell) {
    const component = this.dslCompiler.compile(cell.code, cell.id);

    // Register component
    componentRegistry.register({
      id: cell.id,
      component,
      metadata: { type: 'dsl-generated' }
    });
```

```
    return component;
  }

  // JS evaluation (sandboxed)
  private async evaluateJS(cell: Cell) {
    // Build context from dependencies
    const context = this.buildContext(cell.dependencies);

    // Evaluate with security checks
    const result = await this.jsEvaluator.evaluateCell(cell.id, context);

    // Update state
    this.stateManager.setCellOutput(cell.id, result);

    return result;
  }

  // Build execution context from cell dependencies
  private buildContext(dependencies: string[]): Record<string, any> {
    const context: Record<string, any> = {
      // Core APIs
      data: dataAPI,
      ui: uiAPI,
      state: this.stateManager,

      // Extension APIs
      registerComponent: componentRegistry.register,
      registerCommand: commandRegistry.register,

      // Utility functions
      query: (sql: string) => this.executeQuery(sql),
      fetch: (url: string) => this.secureFetch(url),
    };

    // Add dependency outputs
    for (const depId of dependencies) {
      const output = this.stateManager.getCellOutput(depId);
      context[depId] = output;
    }

    return context;
  }

  // Debounced evaluation for performance
  evaluateDebounced = debounce((cell: Cell) => {
    this.evaluateCell(cell);
  }, 300);
}
```

3.12.6 Performance Optimization

3.12.6.1 Debounced Evaluation

```
// Wait for user to stop typing before evaluating
const debouncedEval = debounce((code: string, cellId: string) => {
  engine.evaluateCell({ id: cellId, code, type: 'javascript', dependencies: [] });
}, 300); // 300ms delay
```

3.12.6.2 Incremental Compilation

```
class IncrementalDSLCompiler {
  compile(code: string, previousAST?: AST): CompiledComponent {
    const newAST = this.parser.parse(code);

    if (previousAST) {
      // Compute minimal diff
      const diff = this.differ.diff(previousAST, newAST);

      // Only recompile changed nodes
      if (diff.changedNodes.length < newAST.nodes.length * 0.3) {
        return this.partialCompile(diff);
      }
    }

    // Full compilation
    return this.fullCompile(newAST);
  }
}
```

3.12.6.3 Virtual Scrolling for Large Outputs

```
// Render only visible cells
import { useVirtualizer } from '@tanstack/react-virtual';

function CellList({ cells }: { cells: Cell[] }) {
  const parentRef = useRef<HTMLDivElement>(null);

  const virtualizer = useVirtualizer({
    count: cells.length,
    getScrollElement: () => parentRef.current,
    estimateSize: () => 200,
  });

  return (
    <div ref={parentRef} style={{ height: '100vh', overflow: 'auto' }}>
      {virtualizer.getVirtualItems().map((virtualRow) => (
        <CellRenderer key={virtualRow.key} cell={cells[virtualRow.index]} />
      ))}
    </div>
  );
}
```

3.12.7 Real-World Pattern Comparison

Platform	DSL Support	JS Evaluation	State Preservation	Dependency Tracking
Observable	No DSL	Live cells	Cell state	Automatic
Evidence	Markdown + Components	Build-time only	Svelte stores	Manual
Count.co	No DSL	SQL + JS	Canvas state	Query deps
tldraw	No DSL	Shape code	History state	Shape deps
Your Framework	Custom DSL	Sandboxed JS	Zustand + HMR	Cell graph

3.12.8 Usage Examples

3.12.8.1 DSL Cell with Live Reload

```
// dashboard.dsl - auto-reloads on save
dashboard "Sales Analytics" {
  layout: grid(3, 2)
  theme: "ocean-blue"

  panel chart {
    id: "revenue-chart"
    type: line
    data: query("SELECT date, revenue FROM sales")
    position: (0, 0, 2, 1)

    options {
      title: "Monthly Revenue"
      colors: ["#0077be", "#00a8e8"]
    }
  }

  panel metric {
    id: "total-revenue"
    data: query("SELECT SUM(revenue) as total FROM sales")
    position: (2, 0, 1, 1)
    format: "currency"
  }
}
```

3.12.8.2 JavaScript Cell with Dependencies

```
// Cell 1: Fetch data
const salesData = await query(`
  SELECT date, revenue, region
  FROM sales
  WHERE date >= '2024-01-01'
`);
```

```
// Cell 2: Transform data (depends on Cell 1)
const aggregated = salesData.reduce((acc, row) => {
  acc[row.region] = (acc[row.region] || 0) + row.revenue;
  return acc;
}, {});

// Cell 3: Visualize (depends on Cell 2)
registerComponent({
  id: 'region-chart',
  component: () => (
    <BarChart data={Object.entries(aggregated).map(([region, revenue]) => ({
      region,
      revenue
    }))} />
  )
});
```

3.12.8.3 Mixed DSL + JS Dashboard

```
// Combine DSL layout with JS logic
const dashboard = {
  // DSL for structure
  layout: compileDSL(`
    dashboard "Interactive Dashboard" {
      layout: grid(2, 2)
      panel container { id: "chart-container", position: (0, 0, 2, 1) }
      panel container { id: "controls", position: (0, 1, 1, 1) }
      panel container { id: "metrics", position: (1, 1, 1, 1) }
    }
  `),

  // JS for interactivity
  cells: [
    {
      id: 'data-fetch',
      type: 'javascript',
      code: 'const data = await fetch("/api/sales").then(r => r.json());'
    },
    {
      id: 'chart-render',
      type: 'javascript',
      dependencies: ['data-fetch'],
      code: 'ui.render("chart-container", <LineChart data={data} />);'
    }
  ]
};
```

3.12.8.4 Error Handling & Recovery

```
// Error boundary for cell evaluation
class CellErrorBoundary extends React.Component<Props, State> {
  state = { hasError: false, error: null };
```

```
  static getDerivedStateFromError(error: Error) {
    return { hasError: true, error };
  }

  componentDidCatch(error: Error, errorInfo: React.ErrorInfo) {
    // Log to error tracking
    console.error('Cell evaluation error:', error, errorInfo);

    // Attempt recovery
    if (this.props.cell.type === 'javascript') {
      // Rollback to last working version
      this.props.onRollback(this.props.cell.id);
    }
  }

  render() {
    if (this.state.hasError) {
      return (
        <ErrorDisplay
          error={this.state.error}
          onRetry={() => this.setState({ hasError: false })}
        />
      );
    }

    return this.props.children;
  }
}
```

For security considerations, see Section 7. For state management, see Section 1.3. For HMR configuration, see Section 2.2.6.

Chapter 4

Security Model

4.1 Overview

A comprehensive security model protects users from malicious extensions while enabling powerful customization capabilities. The security model is based on a combination of sandboxing, permissions, and code review.

4.2 Execution Environment

4.2.1 Sandboxing Approaches

Approach	Description	Pros	Cons	Best For
SES (Secure ECMAScript)	Hardened JavaScript subset	Strong isolation; No global access; Deterministic	Limited APIs; Learning curve; Compatibility	High-security needs
iframe Sandbox	Isolated iframe context	True isolation; Separate origin; CSP support	Communication overhead; Performance; Complex	Untrusted code
Web Workers	Background thread	No DOM access; Isolated; Parallel	No UI; Message passing; Limited	CPU-intensive tasks
Proxy-Based	Intercept API calls	Flexible; Fine-grained; Auditable	Performance overhead; Complex; Bypassable	API control
VM Isolation	Separate JavaScript VM	Complete isolation; Resource limits	Large overhead; Complex; Limited browser support	Maximum security

4.2.2 Sandboxing Library Comparison

Library	Type	Pros	Cons	Use Case
SES (Agoric)	Hardened JS	Strong security; Deterministic; Well-designed	Limited ecosystem; Learning curve	High-security extensions
Realms API	TC39 proposal	Native; Isolated globals; Standard	Not widely supported; Experimental	Future-proof
vm2	Node.js VM	Powerful; Mature	Node-only; Not browser	Server-side only
Sandboxed iframe	Native browser	Built-in; Strong isolation; CSP	Communication overhead; Complex	Untrusted content
Custom Proxy	DIY	Full control; Tailored	Development time; Security risks	Specific needs

4.2.3 BI Dashboard Examples

Platform	Security Model	Implementation
Observable	Runtime sandboxing	• Restricted global scope; • No direct DOM access; • Controlled imports; • Rate limiting
Evidence	Build-time validation	• Component validation; • SQL parameterization; • No runtime eval; • Static analysis
Count.co	SQL sandboxing	• Parameterized queries; • Query validation; • Permission-based access; • Audit logging
tldraw	Client-side validation	• Shape validation; • Canvas bounds checking; • User permissions; • Collaboration security

4.2.4 DOM Access Control

```
// Controlled DOM API
const createSandboxedAPI = (extensionId: string, permissions: string[]) => {
  const allowedAPIs: any = {};

  if (permissions.includes('ui:render')) {
    // Limited DOM access
    allowedAPIs.createElement = (tag: string) => {
      if (!['div', 'span', 'p', 'button'].includes(tag)) {
        throw new Error(`Tag ${tag} not allowed`);
      }
      return document.createElement(tag);
```

```
    };
  }

  if (permissions.includes('storage:local')) {
    // Namespaced storage
    allowedAPIs.storage = {
      get: (key: string) => localStorage.getItem(`ext_${extensionId}_${key}`),
      set: (key: string, value: string) =>
        localStorage.setItem(`ext_${extensionId}_${key}`, value)
    };
  }

  return allowedAPIs;
};
```

For DOM access patterns, see Section 4.1. For API design, see Section 6.3.

4.3 Capability-Based Permissions

4.3.1 Permission Model

Extensions declare required capabilities in manifest:

```
{
  "id": "custom-chart",
  "name": "Custom Chart Extension",
  "version": "1.0.0",
  "permissions": [
    "data:read",
    "ui:render",
    "storage:local"
  ]
}
```

4.3.2 Permission Categories

Permission	Description	Risk Level	Use Cases
`data:read`	Read dashboard data	Low	Visualizations, analytics
`data:write`	Modify dashboard data	Medium	Data transformations, filters
`ui:render`	Render custom UI	Low	Custom components, charts
`storage:local`	Access localStorage	Low	User preferences, cache
`storage:indexed`	Access IndexedDB	Medium	Large datasets, offline
`network:fetch`	Make HTTP requests	High	External APIs, data sources
`network:websocket`	WebSocket connections	High	Real-time data
`system:commands`	Register commands	Low	Custom actions

Permission	Description	Risk Level	Use Cases
`system:keybindings`	Register keybindings	Low	Keyboard shortcuts
`system:eval`	Execute arbitrary code	Critical	Advanced extensions (rarely granted)

4.3.3 Permission Grant Patterns

Pattern	Description	Pros	Cons
Install-Time	User approves on install	Clear; One-time	Users ignore; All-or-nothing
Runtime	Request when needed	Contextual; Granular	Interrupts flow; Frequent prompts
Tiered	Basic vs advanced permissions	Balanced; Progressive	More complex
Automatic	Based on extension type	No prompts; Simple	Less control; Security risk

4.3.4 Runtime Permission Validation

```
class PermissionManager {
  private grants = new Map<string, Set<string>>();

  grant(extensionId: string, permission: string): void {
    if (!this.grants.has(extensionId)) {
      this.grants.set(extensionId, new Set());
    }
    this.grants.get(extensionId)!.add(permission);
    this.audit('grant', extensionId, permission);
  }

  check(extensionId: string, permission: string): boolean {
    return this.grants.get(extensionId)?.has(permission) ?? false;
  }

  enforce(extensionId: string, permission: string): void {
    if (!this.check(extensionId, permission)) {
      this.audit('denied', extensionId, permission);
      throw new Error(`Permission denied: ${permission}`);
    }
  }

  private audit(action: string, extensionId: string, permission: string): void {
    console.log(`[Security] ${action}: ${extensionId} -> ${permission}`);
    // Log to audit system
  }
}
```

For permission UI patterns, see Section 2.2. For audit logging, see Section 6.3.

4.4 API Surface

4.4.1 API Design Principles

- **Minimal Surface**: Only expose necessary functions
- **Explicit Over Implicit**: Clear, documented behavior
- **Versioned**: Maintain backward compatibility
- **Type-Safe**: TypeScript definitions
- **Auditable**: Log all sensitive operations

4.4.2 API Versioning Strategies

Strategy	Description	Pros	Cons
Semantic Versioning	Major.Minor.Patch	Clear; Standard; Predictable	Breaking changes; Migration needed
API Levels	Level 1, 2, 3	Simple; Clear deprecation	Less granular
Feature Flags	Opt-in features	Gradual rollout; A/B testing	Complexity; State explosion
Parallel APIs	v1, v2 coexist	No breaking changes; Smooth migration	Maintenance burden; Code duplication

4.4.3 Audit Logging

```typescript
interface AuditEvent {
  timestamp: number;
  extensionId: string;
  action: string;
  resource: string;
  success: boolean;
  metadata?: Record<string, any>;
}

class AuditLogger {
  private events: AuditEvent[] = [];

  log(event: Omit<AuditEvent, 'timestamp'>): void {
    this.events.push({
      ...event,
      timestamp: Date.now()
    });

    // Persist to IndexedDB
    // Send to analytics
    // Alert on suspicious patterns
  }
```

```
  query(filter: Partial<AuditEvent>): AuditEvent[] {
    return this.events.filter(event =>
      Object.entries(filter).every(([key, value]) =>
        event[key as keyof AuditEvent] === value
      )
    );
  }
}
```

For API design patterns, see Section 1. For versioning, see Section 6.

4.5 Code Review & Signing

4.5.1 Extension Marketplace Security

Layer	Mechanism	Purpose
Submission	Manual review	Human verification
Automated Scan	Static analysis	Detect vulnerabilities
Code Signing	Cryptographic signature	Verify authenticity
Sandboxing	Runtime isolation	Limit damage
Monitoring	Usage analytics	Detect abuse
Reporting	User feedback	Community oversight

4.5.2 Code Signing Implementation

```
// Extension signing
async function signExtension(extensionCode: string, privateKey: CryptoKey): Promise<string> {
  const encoder = new TextEncoder();
  const data = encoder.encode(extensionCode);

  const signature = await crypto.subtle.sign(
    { name: 'RSASSA-PKCS1-v1_5' },
    privateKey,
    data
  );

  return btoa(String.fromCharCode(...new Uint8Array(signature)));
}

// Signature verification
async function verifyExtension(
  extensionCode: string,
  signature: string,
  publicKey: CryptoKey
): Promise<boolean> {
  const encoder = new TextEncoder();
  const data = encoder.encode(extensionCode);
```

```
  const sig = Uint8Array.from(atob(signature), c => c.charCodeAt(0));

  return await crypto.subtle.verify(
    { name: 'RSASSA-PKCS1-v1_5' },
    publicKey,
    sig,
    data
  );
}
```

4.5.3 Security Scanning Tools

Tool	Type	Detects	Use Case
ESLint Security	Static analysis	Common vulnerabilities	Development
npm audit	Dependency scan	Known CVEs	CI/CD
Snyk	Vulnerability DB	Dependencies, code	Production
SonarQube	Code quality	Security hotspots	Enterprise
Custom Rules	Domain-specific	Extension-specific risks	Marketplace

4.5.4 User Warning System

```
interface ExtensionTrust {
  level: 'verified' | 'reviewed' | 'community' | 'unverified';
  badges: string[];
  warnings: string[];
}

function getExtensionTrust(extension: Extension): ExtensionTrust {
  const trust: ExtensionTrust = {
    level: 'unverified',
    badges: [],
    warnings: []
  };

  if (extension.signature && verifySignature(extension)) {
    trust.level = 'verified';
    trust.badges.push('Code Signed');
  }

  if (extension.reviewStatus === 'approved') {
    trust.level = 'reviewed';
    trust.badges.push('Reviewed');
  }

  if (extension.permissions.includes('network:fetch')) {
    trust.warnings.push('Makes external network requests');
  }
```

```
  if (extension.permissions.includes('system:eval')) {
    trust.warnings.push('Can execute arbitrary code');
  }

  return trust;
}
```

4.5.5 Recommended Security Stack

- **Sandboxing**: SES (Secure ECMAScript) for high-security
- **Permissions**: Capability-based with runtime checks
- **API**: Minimal, versioned, type-safe
- **Signing**: Ed25519 or RSA-2048 signatures
- **Scanning**: ESLint Security + Snyk
- **Monitoring**: Audit logs + anomaly detection
- **Marketplace**: Manual review + automated scanning

For plugin architecture, see Section 1.4. For extension development, see Section 2.1.

> **Note**: Security technologies (SES, CSP, Subresource Integrity) are detailed in Section 6.3.

Chapter 5

Advanced Features

Modern BI dashboard capabilities including canvas interfaces, SQL integration, real-time collaboration, and performance optimization.

5.1 Canvas Architecture

5.1.1 Overview

Canvas-based interfaces (inspired by Count.co and tldraw) provide infinite workspace for flexible data exploration and visualization.

5.1.2 Infinite Canvas Pattern

Aspect	Implementation	Pros	Cons
Viewport Management	Transform matrix	Smooth pan/zoom; Efficient rendering	Complex math; Coordinate transforms
Virtualization	Render visible area only	Performance; Scales to large canvases	Implementation complexity; Edge cases
Spatial Indexing	R-tree or Quadtree	Fast queries; Collision detection	Memory overhead; Update complexity
Layer System	Separate canvas layers	Compositing; Selective updates	More canvases; Coordination

5.1.3 Cell/Shape Positioning Systems

System	Description	Use Case
Absolute Positioning	Fixed x, y coordinates	Manual layout, precise control
Relative Positioning	Position relative to parent	Nested components, groups
Auto-Layout	Algorithm-based positioning	Automatic organization, graphs

System	Description	Use Case
Grid System	Snap-to-grid alignment	Structured dashboards
Flow Layout	Flexbox/Grid-like	Responsive arrangements

5.1.4 Rendering Strategies

```
// Canvas rendering with virtualization
class CanvasRenderer {
  private viewport: Viewport;
  private spatialIndex: RTree;

  render(ctx: CanvasRenderingContext2D) {
    // Get visible bounds
    const bounds = this.viewport.getVisibleBounds();

    // Query spatial index for visible items
    const visibleItems = this.spatialIndex.query(bounds);

    // Render only visible items
    for (const item of visibleItems) {
      this.renderItem(ctx, item);
    }
  }

  renderItem(ctx: CanvasRenderingContext2D, item: CanvasItem) {
    ctx.save();

    // Apply viewport transform
    this.viewport.applyTransform(ctx);

    // Render item
    item.render(ctx);

    ctx.restore();
  }
}
```

5.1.5 Platform Examples

Platform	Canvas Implementation	Key Features
tldraw	Custom canvas engine	• Infinite canvas; • Shape system; • Collaborative cursors; • History/undo
Count.co	React + Canvas	• Cell-based layout; • Free-form positioning; • Auto-layout options; • SQL-driven cells

Platform	Canvas Implementation	Key Features
Observable	HTML/SVG cells	• Linear notebook flow; • Custom layouts via HTML; • D3.js integration

5.1.6 Recommended Stack

- **Canvas Library**: Konva.js, Fabric.js, or custom WebGL
- **Spatial Index**: rbush (R-tree implementation)
- **Transform**: gl-matrix or custom matrix math
- **Gestures**: Hammer.js or custom touch handling

For state management, see Section 1.3. For collaboration, see Section 4.3.

5.2 SQL Integration

5.2.1 Overview

SQL-driven dashboards (inspired by Count.co) enable powerful data analysis with familiar query syntax.

5.2.2 Query Execution Architecture

Approach	Implementation	Pros	Cons
Client-Side SQL	DuckDB WASM, SQLite WASM	No backend needed; Fast queries; Offline capable	Large bundle; Memory limits; Initial load time
Server-Side SQL	PostgreSQL, MySQL	Unlimited data; Mature ecosystem; Security	Network latency; Backend required; Scaling costs
Hybrid	Cache + server	Best of both; Optimized	Complexity; Sync issues

5.2.3 DuckDB WASM Architecture

```
// DuckDB WASM integration
import * as duckdb from '@duckdb/duckdb-wasm';

class SQLEngine {
  private db: duckdb.AsyncDuckDB;
  private conn: duckdb.AsyncDuckDBConnection;

  async initialize() {
    const bundle = await duckdb.selectBundle({
      mvp: {
        mainModule: duckdb_wasm,
        mainWorker: duckdb_wasm_worker
      }
```

```
    });

    const worker = new Worker(bundle.mainWorker!);
    const logger = new duckdb.ConsoleLogger();
    this.db = new duckdb.AsyncDuckDB(logger, worker);
    await this.db.instantiate(bundle.mainModule);
    this.conn = await this.db.connect();
  }

  async query(sql: string, params?: any[]) {
    // Parameterized query for security
    const stmt = await this.conn.prepare(sql);
    const result = await stmt.query(...(params || []));
    return result.toArray();
  }

  async loadData(tableName: string, data: any[]) {
    // Load data into DuckDB
    await this.conn.insertArrowTable(
      tableName,
      arrowTable(data)
    );
  }
}
```

5.2.4 Query Result Caching

Strategy	Implementation	Use Case
In-Memory Cache	Map/LRU cache	Fast repeated queries
IndexedDB Cache	Persistent cache	Large result sets
Query Fingerprint	Hash-based key	Cache invalidation
Incremental Updates	Delta queries	Real-time data

5.2.5 Data Binding Patterns

```
// Reactive SQL queries
import { useQuery } from './sql-hooks';

function DataCell({ sql, params }) {
  const { data, loading, error, refetch } = useQuery(sql, params);

  // Automatically re-execute when params change
  useEffect(() => {
    refetch();
  }, [params]);

  if (loading) return <Spinner />;
  if (error) return <Error message={error.message} />;
```

```
  return <DataTable data={data} />;
}
```

5.2.6 Platform Examples

Platform	SQL Implementation	Features
Count.co	DuckDB + PostgreSQL	• SQL cells; • Query dependencies; • Result caching; • Parameterized queries
Observable	SQL cells (via connectors)	• Database connectors; • SQL template literals; • Reactive queries
Evidence	DuckDB + connectors	• SQL + Markdown; • Component binding; • Build-time queries

5.2.7 Recommended Stack

- **Client SQL**: DuckDB WASM (analytics), SQLite WASM (simple queries)
- **Caching**: React Query or SWR with IndexedDB
- **Parameterization**: Prepared statements, tagged templates
- **Visualization**: Observable Plot, Vega-Lite

For data persistence, see Section 9.3. For component binding, see Section 4.1.

5.3 Real-Time Collaboration

5.3.1 Overview

Multi-user collaboration (inspired by Count.co and tldraw) enables teams to work together in real-time.

5.3.2 CRDT (Conflict-free Replicated Data Type) Libraries

Library	Language	Pros	Cons	Bundle Size	Use Case
Yjs	JavaScript	Mature; Performant; Rich ecosystem; CRDT types	Learning curve; Bundle size	~50KB	Production apps
Automerge	JavaScript/Rust	Pure CRDT; Offline-first; Time travel	Larger bundle; Performance	~200KB	Offline-first apps

Library	Language	Pros	Cons	Bundle Size	Use Case
Loro	Rust (WASM)	High performance; Small bundle; Rich text	New/experimental; Smaller ecosystem	~100KB	Performance-critical
Fluid Framework	TypeScript	Microsoft-backed; Enterprise features	Complex; Azure dependency	Large	Enterprise

5.3.3 Yjs Integration Architecture

```
// Yjs collaborative state
import * as Y from 'yjs';
import { WebrtcProvider } from 'y-webrtc';
import { IndexeddbPersistence } from 'y-indexeddb';

class CollaborationEngine {
  private doc: Y.Doc;
  private provider: WebrtcProvider;
  private persistence: IndexeddbPersistence;

  constructor(roomId: string) {
    this.doc = new Y.Doc();

    // WebRTC provider for P2P sync
    this.provider = new WebrtcProvider(roomId, this.doc);

    // IndexedDB for offline persistence
    this.persistence = new IndexeddbPersistence(roomId, this.doc);
  }

  // Shared canvas state
  getCanvas(): Y.Map<any> {
    return this.doc.getMap('canvas');
  }

  // Shared cells/shapes
  getCells(): Y.Array<any> {
    return this.doc.getArray('cells');
  }

  // Awareness (presence)
  getAwareness() {
    return this.provider.awareness;
  }
}
```

5.3.4 Presence System

Feature	Implementation	Use Case
User Cursors	Awareness state + SVG	Show where users are pointing
Active Selection	Highlighted shapes/cells	Show what users are editing
User Avatars	Avatar component	Identify collaborators
Activity Feed	Event log	Show recent changes
Typing Indicators	Awareness + debounce	Show who's typing

5.3.5 Conflict Resolution Strategies

Strategy	Description	Pros	Cons
Last-Write-Wins	Timestamp-based	Simple; Fast	Data loss; Not fair
CRDT	Mathematically convergent	No conflicts; Automatic	Complex; Memory overhead
Operational Transform	Transform operations	Proven; Google Docs uses it	Very complex; Hard to implement
Manual Resolution	User chooses	User control; Transparent	Interrupts flow; User burden

5.3.6 Platform Examples

Platform	Collaboration System	Implementation
tldraw	Yjs + WebRTC	• Real-time shape sync; • Presence cursors; • History preservation; • Offline support
Count.co	Custom WebSocket	• Real-time cell updates; • Collaborative editing; • Presence indicators; • Comment threads
Observable	Limited collaboration	• Notebook sharing; • Fork-based workflow; • No real-time sync

5.3.7 Recommended Architecture

```
// Collaborative canvas cell
function CollaborativeCell({ cellId }) {
  const collab = useCollaboration();
  const cells = collab.getCells();

  // Subscribe to cell changes
  const cell = useYArray(cells, cellId);

  // Update cell
```

```
  const updateCell = (changes) => {
    collab.doc.transact(() => {
      const cellData = cells.get(cellId);
      Object.assign(cellData, changes);
    });
  };

  // Show presence
  const awareness = collab.getAwareness();
  const users = useAwareness(awareness);

  return (
    <Cell data={cell} onChange={updateCell}>
      <PresenceCursors users={users} />
    </Cell>
  );
}
```

5.3.8 Recommended Stack

- **CRDT**: Yjs (most mature and performant)
- **Transport**: WebRTC (P2P) or WebSocket (server-based)
- **Persistence**: IndexedDB (offline) + server backup
- **Presence**: Yjs Awareness API
- **Cursors**: Custom SVG overlay

For state management, see Section 1.3. For canvas architecture, see Section 4.1.

5.4 Performance Optimization

5.4.1 Overview

Canvas-based and data-intensive applications require specific performance optimizations.

5.4.2 Canvas Rendering Optimizations

Technique	Description	Performance Gain	Complexity
Virtual Scrolling	Render visible items only	10-100x	Medium
Layer Separation	Multiple canvas layers	2-5x	Low
WebGL Rendering	GPU-accelerated	10-50x	High
Offscreen Canvas	Background rendering	2-3x	Medium
Request Animation Frame	Batch updates	2-5x	Low
Dirty Rectangle	Partial redraws	5-10x	Medium

5.4.3 State Management Optimizations

```
// Structural sharing with Immer
import { produce } from 'immer';
```

```
const updateCanvas = produce((draft, action) => {
  // Immer creates efficient immutable updates
  const cell = draft.cells.find(c => c.id === action.cellId);
  if (cell) {
    cell.position = action.position;
  }
});

// Memoization for expensive computations
import { useMemo } from 'react';

function DataVisualization({ data, config }) {
  const processedData = useMemo(() => {
    // Expensive data transformation
    return processLargeDataset(data, config);
  }, [data, config]);

  return <Chart data={processedData} />;
}

// Lazy computation with computed values
import { computed } from '@preact/signals-react';

const visibleCells = computed(() => {
  const viewport = canvasState.viewport.value;
  return canvasState.cells.value.filter(cell =>
    isInViewport(cell, viewport)
  );
});
```

5.4.4 Data Loading Strategies

Strategy	Implementation	Use Case
Lazy Loading	Load on demand	Large datasets
Pagination	Load in chunks	Infinite scroll
Streaming	Progressive loading	Real-time data
Prefetching	Load ahead	Predictable navigation
Caching	Store results	Repeated queries

5.4.5 Bundle Optimization

```
// Code splitting for large features
const AdvancedChart = lazy(() => import('./AdvancedChart'));

// Tree shaking - import only what you need
import { map, filter } from 'lodash-es';
```

```javascript
// Dynamic imports for optional features
async function loadCollaboration() {
  if (user.hasPremium) {
    const { CollaborationEngine } = await import('./collaboration');
    return new CollaborationEngine();
  }
}
```

5.4.6 Platform Benchmarks

Platform	Initial Load	Canvas Render	Query Execution
tldraw	~200KB	60 FPS (1000 shapes)	N/A
Count.co	~500KB	60 FPS (100 cells)	<100ms (DuckDB)
Observable	~300KB	60 FPS (cells)	Varies

5.4.7 Recommended Optimizations

1. **Rendering**: Virtual scrolling + layer separation
2. **State**: Immer for immutability + signals for reactivity
3. **Data**: DuckDB WASM + IndexedDB caching
4. **Bundle**: Code splitting + tree shaking
5. **Network**: Service worker + prefetching

For canvas rendering, see Section 4.1. For state management, see Section 1.3.

Chapter 6

Data Persistence

Configuration and data persistence strategies for the framework.

6.1 Storage Strategy & Configuration

6.2 User Configurations

6.3 Settings Management

6.3.1 Concept

Centralized system for user preferences and application options.

6.3.2 Architecture Approaches

1. **Hierarchical Settings Structure**
   - Nested configuration organized by domain (general, dashboard, visualization, performance)
   - Supports inheritance and overrides at different levels
   - Type-safe with schema validation
2. **Flat Key-Value Store**
   - Simple key-value pairs with dot notation (e.g., `dashboard.refreshInterval`)
   - Easy to serialize and persist
   - Less structure, more flexibility
3. **Hybrid Approach**
   - Hierarchical structure in memory
   - Flat storage for persistence
   - Best of both worlds

6.3.3 Pros & Cons Analysis

Approach	Pros	Cons	Best For
Hierarchical	Clear organization; Type safety; Easy validation; Supports overrides	More complex to implement; Harder to query dynamically; Deeper nesting complexity	Large applications with many settings categories
Flat Key-Value	Simple implementation; Easy persistence; Dynamic queries; Minimal overhead	No structure enforcement; Harder to validate; No type safety; Namespace collisions	Small to medium apps, simple preferences
Hybrid	Structured in code; Simple persistence; Type-safe + flexible; Best performance	Transformation overhead; Two representations to maintain; More complex architecture	Production BI dashboards requiring both structure and flexibility

6.3.4 State Management Library Comparison

Library	Architecture	Pros	Cons	Use Case
Zustand	Flux-like store	Simple API; No providers; Middleware ecosystem; Small bundle	Manual optimization needed; Global state only	General-purpose settings management
Jotai	Atomic state	Fine-grained reactivity; Minimal re-renders; Composable atoms; Bottom-up	Learning curve; More boilerplate; Debugging complexity	Complex, interconnected settings
Valtio	Proxy-based	Mutable API; Automatic tracking; Minimal boilerplate; Intuitive	Proxy limitations; Debugging harder; Less ecosystem	Rapid development, simple state
Redux Toolkit	Redux pattern	Mature ecosystem; DevTools; Predictable; Time-travel	Verbose; Boilerplate; Learning curve; Larger bundle	Enterprise apps, complex workflows

6.3.5 Schema Validation Approaches

Approach	Pros	Cons
Runtime Validation (Zod, Yup)	Catches invalid data; User input protection; Type inference; Clear error messages	Runtime overhead; Bundle size increase; Validation logic duplication
TypeScript Only	Zero runtime cost; Compile-time safety; No bundle impact; IDE support	No runtime protection; Can't validate external data; Type erasure at runtime
Hybrid (TS + Runtime)	Best safety; Validates external data; Type-safe in code; Comprehensive	Maintenance overhead; Schema duplication; Larger bundle

6.3.6 Persistence Strategy Comparison

Strategy	Pros	Cons	Recommended For
Eager Persistence (Save on every change)	No data loss; Always in sync; Simple logic	Performance overhead; Excessive writes; Storage wear	Critical settings, small config
Debounced Persistence (Save after delay)	Reduced writes; Better performance; Batched updates	Potential data loss; Complexity; Timing issues	Frequently changing settings
Manual Persistence (Save on action)	User control; Minimal writes; Predictable	User must remember; Data loss risk; Poor UX	Power user tools, explicit saves
Hybrid (Critical eager, others debounced)	Balanced approach; Optimized performance; Data safety	Complex logic; More code; Configuration needed	Production BI dashboards

6.3.7 Recommended Architecture

Hierarchical structure with Zustand, Zod validation, hybrid persistence (critical settings eager, UI preferences debounced), IndexedDB storage.

6.4 Keybinding System

6.4.1 Concept

Customizable keyboard shortcuts for commands and actions.

6.4.2 Architecture Approaches

1. **Command-Based Architecture**
   - Commands are first-class entities with IDs, names, and execution logic

- Keybindings map to commands (many-to-one relationship)
- Context-aware execution with "when" clauses
- Supports command palette and keybinding customization

2. **Direct Event Binding**
   - Keybindings directly trigger functions
   - No command abstraction layer
   - Simpler but less flexible
3. **Keymap Hierarchy**
   - Global keybindings (always active)
   - Mode-specific keybindings (context-aware)
   - Component-local keybindings (scoped)
   - Priority-based resolution

6.4.3 Pros & Cons Analysis

Approach	Pros	Cons	Best For
Command-Based	Decoupled commands from keys; Easy customization; Command palette support; Context-aware execution; Discoverable	More complex architecture; Additional abstraction layer; Higher memory overhead; Steeper learning curve	Complex applications with many commands, power users
Direct Binding	Simple implementation; Minimal overhead; Easy to understand; Fast execution	Hard to customize; No command palette; Tight coupling; Difficult to document	Simple apps, fixed keybindings, prototypes
Keymap Hierarchy	Context-aware; Scoped bindings; Priority resolution; Flexible	Complexity in resolution; Potential conflicts; Debugging difficulty; State management	Multi-mode applications, context-sensitive UIs

6.4.4 Keybinding Library Comparison

Library	Size	Pros	Cons	Use Case
tinykeys	400B	Minimal size; Zero dependencies; Chord support; Modern API	Limited features; No scope support; Manual context handling	Size-constrained apps, simple keybindings

Library	Size	Pros	Cons	Use Case
hotkeys-js	~3KB	Scope support; Key filtering; Feature-rich; Mature	Larger bundle; Older API style; Less TypeScript support	General-purpose, scope-aware bindings
Mousetrap	~2KB	Popular; Well-documented; Chord sequences; Mature	Not actively maintained; No TypeScript; Older patterns	Legacy apps, proven stability
react-hotkeys-hook	~2KB	React hooks; Component-scoped; TypeScript support; Modern	React-only; Re-render considerations; Hook limitations	React applications, component-local bindings
Custom Solution	Varies	Full control; Tailored features; No dependencies; Optimized	Development time; Maintenance burden; Testing overhead; Edge cases	Unique requirements, full control needed

6.4.5 Key Conflict Resolution Strategies

Strategy	Pros	Cons
Priority-Based (Global → Mode → Local)	Clear hierarchy; Predictable; Easy to reason about	May override important globals; Inflexible; Can't express complex rules
Context-Aware (When clauses)	Flexible; Expressive; Handles complex cases; Fine-grained control	Complex to implement; Harder to debug; Performance overhead
User-Defined Priority	User control; Flexible; Handles edge cases	Complex UI; User confusion; Maintenance burden

6.4.6 Chord Sequence Considerations

Aspect	Pros	Cons
Multi-Key Sequences (e.g., Ctrl+K Ctrl+S)	More key combinations; Familiar to power users; Namespace expansion	Discoverability issues; Timing complexity; Harder for beginners
Single Keys Only	Simple; Fast; Easy to learn	Limited combinations; Conflicts more likely; Less powerful

6.4.7 Recommended Architecture

Command-based with keymap hierarchy, context-aware execution, chord support, and user customization. Use tinykeys for minimal apps, hotkeys-js for feature-rich needs.

6.5 Theme System

6.5.1 Concept

Visual styling and color schemes that can be switched dynamically.

6.5.2 Architecture Approaches

1. **CSS Variables Approach**
   - Define theme tokens as CSS custom properties
   - Switch themes by changing root-level variables
   - No JavaScript required for styling
   - Native browser support
2. **CSS-in-JS with Theme Context**
   - Theme object passed through React context
   - Styles generated at runtime
   - Full JavaScript access to theme values
   - Dynamic styling capabilities
3. **Build-Time Theme Generation**
   - Themes compiled to separate CSS files
   - Zero runtime overhead
   - Static theme switching
   - Optimal performance
4. **Hybrid Approach**
   - CSS variables for colors and tokens
   - CSS-in-JS for complex dynamic styles
   - Best of both worlds

6.5.3 Pros & Cons Analysis

Approach	Pros	Cons	Best For
CSS Variables	Zero runtime cost; Native browser support; Simple implementation; No JavaScript needed; Excellent performance	Limited browser support (old browsers); No complex logic; String values only; Less type safety	Modern browsers, performance-critical apps
CSS-in-JS (Runtime)	Full JavaScript access; Dynamic styling; Type-safe; Component-scoped; Conditional styles	Runtime overhead; Larger bundle; FOUC potential; Performance impact; Hydration issues	Complex theming, dynamic styles

Approach	Pros	Cons	Best For
Build-Time	Zero runtime cost; Optimal performance; Static analysis; Type-safe; Small bundle	No runtime switching; Build complexity; Less flexible; Requires rebuild	Static themes, maximum performance
Hybrid	Balanced performance; Flexible; Type-safe; Best of both	More complex; Two systems to maintain; Learning curve	Production BI dashboards

6.5.4 Styling Library Comparison

Library	Approach	Pros	Cons	Use Case
Tailwind CSS	Utility-first	Rapid development; Small production bundle; Dark mode built-in; Consistent design	Verbose HTML; Learning curve; Customization limits; Not semantic	Fast development, consistent UI
Styled-Components	Runtime CSS-in-JS	Component-scoped; Dynamic theming; Popular ecosystem; TypeScript support	Runtime overhead; Bundle size; SSR complexity; Performance cost	Dynamic theming, component libraries
Stitches	Near-zero runtime	Minimal runtime; Variants API; Type-safe; Good performance	Smaller ecosystem; Learning curve; Less mature	Performance + flexibility balance
vanilla-extract	Build-time	Zero runtime; Type-safe; Best performance; CSS Modules-like	Build complexity; No runtime theming; Smaller ecosystem	Maximum performance, static themes

Library	Approach	Pros	Cons	Use Case
Emotion	Runtime CSS-in-JS	Flexible; Framework-agnostic; Good performance; Popular	Runtime cost; Bundle size; Complexity	Framework-agnostic, flexible theming
CSS Modules	Build-time	Simple; Scoped styles; Zero runtime; Familiar CSS	No dynamic theming; Verbose; Limited features	Simple apps, traditional CSS

6.5.5 Theme Switching Strategies

Strategy	Pros	Cons
Class-Based (`<html class="dark">`)	Simple; CSS-only; Fast; No FOUC	Limited to predefined themes; No gradual transitions
Attribute-Based (`<html data-theme="dark">`)	Semantic; Multiple themes; CSS-only; Accessible	Slightly more verbose; Browser support
Context-Based (React Context)	JavaScript access; Dynamic values; Type-safe	Runtime overhead; Re-render cost; Complexity
CSS Variable Injection	Dynamic; Performant; Flexible	JavaScript required; FOUC potential

6.5.6 System Preference Integration

Aspect	Pros	Cons
Auto-Detect (`prefers-color-scheme`)	Respects user preference; Better UX; Accessibility; Native API	User can't override easily; May not match app context
Manual Selection	User control; Predictable; Simple	Ignores system preference; Extra UI needed
Hybrid (Auto + Manual Override)	Best UX; Respects preference; User control	More complex; State management needed

6.5.7 Recommended Architecture

CSS Variables for tokens, Tailwind CSS for utility classes, system preference detection with manual override, persistent user choice in IndexedDB.

6.6 Layout System

6.6.1 Concept

Flexible, user-customizable arrangement of dashboard components and panels.

6.6.2 Architecture Patterns

1. **Grid-Based Layout**

```
interface GridLayout {
  id: string;
  items: GridItem[];
  cols: number;
  rowHeight: number;
}

interface GridItem {
  id: string;
  x: number;
  y: number;
  w: number;
  h: number;
  component: string;
  props: Record<string, any>;
}
```

2. **Split Pane Layout** (Recursive)

```
interface SplitLayout {
  type: 'horizontal' | 'vertical';
  children: (SplitLayout | PanelLayout)[];
  sizes: number[]; // Percentage splits
}

interface PanelLayout {
  type: 'panel';
  component: string;
  props: Record<string, any>;
}
```

6.6.3 Best Libraries & Tools

1. **react-grid-layout** (Most popular)
   - Drag-and-drop grid
   - Responsive breakpoints
   - Collision detection
   - Used by: Grafana, many BI dashboards

```
import GridLayout from 'react-grid-layout';

<GridLayout
  layout={layout}
  cols={12}
  rowHeight={30}
  onLayoutChange={handleLayoutChange}
  draggableHandle=".drag-handle"
>
  {items.map(item => <div key={item.id}>{item.content}</div>)}
```

```
</GridLayout>
```

2. **react-mosaic** (Split pane layouts)
   - Nested split views
   - Drag-and-drop rearrangement
   - Used by: Code editors, complex dashboards

```
import { Mosaic } from 'react-mosaic-component';

<Mosaic
  value={mosaicLayout}
  onChange={setMosaicLayout}
  renderTile={(id) => <Panel id={id} />}
/>
```

3. **golden-layout** (Advanced)
   - Multi-window support
   - Tab containers
   - Popout windows
   - Used by: Trading platforms, complex BI tools
4. **allotment** (VS Code-style)
   - Split panes with resizable dividers
   - Nested layouts
   - Keyboard accessible
   - Used by: Code-like interfaces
5. **react-resizable-panels** (Modern)
   - Lightweight, accessible
   - Imperative API
   - Persistent layouts
   - Used by: Modern React apps

6.6.4 BI Dashboard Examples

- **Observable**: Notebook-style (vertical flow) + custom layouts
- **Evidence**: Page-based layouts with component slots
- **Grafana**: react-grid-layout for dashboard panels
- **Metabase**: Fixed grid with responsive breakpoints
- **Apache Superset**: react-grid-layout with custom extensions
- **Tableau**: Proprietary grid system with containers

6.6.5 Implementation Pattern

```
// Layout manager with persistence
import { create } from 'zustand';
import { persist } from 'zustand/middleware';

interface LayoutStore {
  layouts: Record<string, Layout>;
  activeLayout: string;
  saveLayout: (id: string, layout: Layout) => void;
  loadLayout: (id: string) => void;
```

```
  deleteLayout: (id: string) => void;
}

const useLayoutStore = create<LayoutStore>()(
  persist(
    (set, get) => ({
      layouts: {},
      activeLayout: 'default',

      saveLayout: (id, layout) =>
        set((state) => ({
          layouts: { ...state.layouts, [id]: layout }
        })),

      loadLayout: (id) => {
        const layout = get().layouts[id];
        if (layout) {
          set({ activeLayout: id });
          applyLayout(layout);
        }
      },

      deleteLayout: (id) =>
        set((state) => {
          const { [id]: _, ...rest } = state.layouts;
          return { layouts: rest };
        })
    }),
    { name: 'dashboard-layouts' }
  )
);

// Layout presets
const layoutPresets = {
  default: {
    type: 'grid',
    items: [/* ... */]
  },
  analytics: {
    type: 'split',
    direction: 'horizontal',
    children: [/* ... */]
  },
  monitoring: {
    type: 'grid',
    items: [/* ... */]
  }
};
```

6.6.6 Advanced Layout Features

1. **Responsive Breakpoints**

```
interface ResponsiveLayout {
  lg: GridItem[]; // Desktop
  md: GridItem[]; // Tablet
  sm: GridItem[]; // Mobile
}
```

2. **Layout Templates**

   - Predefined layouts for common use cases
   - One-click application
   - Customizable after application

3. **Layout Sharing**

   - Export layout as JSON
   - Import from URL or file
   - Team templates

4. **Layout History**

   - Undo/redo support
   - Version history
   - Restore previous layouts

6.6.7 Recommended Architecture for BI Dashboards

```
// Unified configuration system
interface DashboardConfig {
  version: string;
  settings: Settings;
  keybindings: KeybindingConfig;
  theme: ThemeConfig;
  layout: LayoutConfig;
  extensions: ExtensionConfig[];
}

// Configuration manager
class ConfigManager {
  private config: DashboardConfig;
  private storage: StorageAdapter;
  private listeners: Set<(config: DashboardConfig) => void>;

  async load(): Promise<DashboardConfig> {
    const stored = await this.storage.get('dashboard-config');
    this.config = stored || defaultConfig;
    return this.config;
  }

  async save(): Promise<void> {
    await this.storage.set('dashboard-config', this.config);
    this.notifyListeners();
```

```
  }

  update(path: string, value: any): void {
    set(this.config, path, value);
    this.save();
  }

  export(): string {
    return JSON.stringify(this.config, null, 2);
  }

  import(json: string): void {
    const imported = JSON.parse(json);
    this.config = migrateConfig(imported);
    this.save();
  }
}
```

6.6.8 Best Practices from Leading BI Platforms

1. **Observable**:
   - Reactive configuration (changes propagate automatically)
   - Notebook-level and cell-level settings
   - Git-friendly (text-based configs)
2. **Evidence**:
   - YAML for project config (version controlled)
   - UI for user preferences (browser storage)
   - Environment-based overrides
3. **Grafana**:
   - Hierarchical settings (global → org → dashboard → panel)
   - JSON-based dashboard definitions
   - Plugin-extensible configuration
4. **Metabase**:
   - Database-backed configuration
   - Admin UI for system settings
   - User preferences in browser storage

6.6.9 Recommended Stack for Your Framework

```
// Settings: Zustand + Zod + IndexedDB
// Keybindings: tinykeys + custom registry
// Themes: Tailwind CSS + CSS variables
// Layouts: react-grid-layout + react-resizable-panels
// Persistence: IndexedDB with migration support
// Export/Import: JSON with schema validation
```

6.7 Extension State

- **Plugin Configurations**: Extension-specific settings

- **Installed Extensions**: List of active plugins
- **Extension Data**: Plugin-managed data

6.8 Storage Options & Strategy

6.8.1 Client-Side Storage

6.8.2 Primary Storage

- **IndexedDB**: Large, structured data (dashboards, datasets, extension state)
  - Capacity: 50MB+ (typically unlimited with user prompt at ~50MB, can reach GBs)
  - Advantages: Large capacity, structured queries, transactions, async API
  - Use Case: Main persistent storage for dashboards and configurations
- **LocalStorage**: Small, simple key-value pairs (preferences)
  - Capacity: 5-10MB (varies by browser)
  - Advantages: Simple API, synchronous access
  - Limitations: Small size limit, string-only storage, synchronous (blocks UI)
  - Use Case: Basic preferences, feature flags

6.8.3 Advanced Client Storage

- **Cache API**: HTTP responses, assets, and API data
  - Capacity: Similar to IndexedDB (typically unlimited with prompt)
  - Advantages: Built for PWAs, offline-first, versioned caches
  - Use Case: Dashboard templates, static assets, API response caching
- **OPFS (Origin Private File System)**: High-performance file operations
  - Capacity: Large (GBs, quota-managed like IndexedDB)
  - Advantages: Fast, large capacity, works with Web Workers, better performance
  - Use Case: Large dataset caching, temporary file operations
- **File System Access API**: Direct file system read/write
  - Capacity: Limited only by user's disk space
  - Advantages: Native file integration, user control, large files
  - Limitations: Requires user permission, limited browser support
  - Use Case: Export/import dashboard configs, large dataset files

6.8.4 Memory-Based Storage

- **In-Memory State (Zustand/Jotai)**: Session-only volatile state
  - Capacity: Limited by browser's available RAM (typically hundreds of MBs)
  - Advantages: Fastest access, no serialization overhead
  - Limitations: Lost on page refresh, memory-constrained
  - Use Case: Active dashboard state, UI state, temporary calculations
- **SessionStorage**: Tab-scoped temporary data
  - Capacity: 5-10MB (same as LocalStorage)
  - Advantages: Automatic cleanup on tab close
  - Limitations: Small size, string-only storage
  - Use Case: Temporary filters, session-specific preferences

6.8.5 Server-Side/Hybrid Storage

6.8.5.1 Backend Integration

- **REST/GraphQL API**: Centralized data storage
  - Capacity: Unlimited (depends on server infrastructure)
  - Advantages: Scalable, secure, multi-user collaboration
  - Use Case: User accounts, shared dashboards, enterprise deployments
- **Firebase/Supabase**: Managed backend services
  - Capacity: Varies by plan (Free: 1GB, Paid: scalable to TBs)
  - Advantages: Real-time sync, built-in auth, managed infrastructure
  - Limitations: Vendor lock-in, cost at scale
  - Use Case: Rapid prototyping, real-time collaboration
- **PouchDB + CouchDB**: Offline-first with server sync
  - Capacity: Client (IndexedDB limits), Server (unlimited)
  - Advantages: Automatic sync, conflict resolution, works offline
  - Use Case: Offline-capable dashboards with eventual consistency

6.8.5.2 Peer-to-Peer

- **WebRTC Data Channels**: Direct peer-to-peer data sharing
  - Capacity: Limited by network bandwidth (not storage-based)
  - Advantages: No server required, direct user-to-user sync
  - Limitations: Complex setup, requires signaling server, ephemeral
  - Use Case: Collaborative editing without central server

6.8.6 Specialized Storage

6.8.6.1 Database Engines

- **SQLite WASM (sql.js)**: Full SQL database in browser
  - Capacity: Limited by available RAM (typically 100s of MBs)
  - Advantages: SQL queries, relational data, transactions
  - Limitations: Entire DB in memory, manual persistence to IndexedDB
  - Use Case: Complex queries on dashboard data, analytics
- **DuckDB WASM**: Analytical queries on large datasets
  - Capacity: Can handle GBs of data (with streaming/chunking)
  - Advantages: OLAP queries, Parquet support, fast analytics, columnar storage
  - Use Case: In-browser data analytics, large dataset processing
- **RxDB**: Reactive, offline-first database
  - Capacity: Uses IndexedDB underneath (50MB+ with prompt)
  - Advantages: Observable queries, multi-tab sync, encryption
  - Use Case: Reactive dashboards, multi-tab coordination

6.8.6.2 Decentralized Storage

- **Gun.js**: Decentralized graph database
  - Capacity: Limited by IndexedDB locally, distributed across peers
  - Advantages: P2P sync, offline-first, decentralized
  - Use Case: Decentralized apps, graph-based data
- **IPFS**: Distributed file storage
  - Capacity: Unlimited (distributed across network), local cache limited

  - Advantages: Content-addressed, permanent, decentralized
  - Limitations: Requires gateway/node, slower access
  - Use Case: Immutable dashboard templates, public data sharing
- **Ceramic Network**: Decentralized data with DIDs
  - Capacity: Unlimited (distributed), per-stream limits vary
  - Advantages: User-owned data, cross-app portability
  - Limitations: Emerging tech, requires infrastructure
  - Use Case: User-owned dashboard configurations

6.8.7 Recommended Tiered Strategy

Tier 1: Hot Data (Active) - In-Memory State → Current dashboard state, UI state - SessionStorage → Temporary filters, session data

Tier 2: Warm Data (Recent) - IndexedDB → Dashboards, extensions, user preferences - Cache API → Dashboard templates, static resources

Tier 3: Cold Data (Archive) - Cloud Storage/Backend API → Backups, shared dashboards - File System Access → Export/import, large files

Tier 4: Analytics (Optional) - DuckDB WASM → In-browser analytics on large datasets - SQLite WASM → Complex relational queries

6.8.8 Comparative Analysis: Pros & Cons

Storage Type	Pros	Cons	Best For
IndexedDB	Large capacity (GBs); Structured queries; Transactions; Async (non-blocking); Wide browser support	Complex API; No cross-origin access; User can clear data; Quota management needed	Primary persistent storage for dashboards, configs, extension data
LocalStorage	Simple API; Synchronous access; Wide browser support; Easy to use	Small capacity (5-10MB); String-only storage; Synchronous (blocks UI); No transactions	Small preferences, feature flags, simple settings
Cache API	Built for offline-first; Version control; Large capacity; Perfect for assets	Not for structured data; More complex than LocalStorage; Requires service worker knowledge	Static assets, API responses, dashboard templates
OPFS	High performance; Large capacity (GBs); Works with Workers; File-based operations	Limited browser support; Newer API (less mature); More complex API; Not for small data	Large datasets, file operations, high-performance needs

Storage Type	Pros	Cons	Best For
File System Access	Unlimited capacity; Native file integration; User control; Cross-app sharing	Requires user permission; Limited browser support; Security restrictions; Not automatic	Import/export, user-managed backups, large files
In-Memory State	Fastest access; No serialization; Simple to use; No quota limits	Lost on refresh; RAM-limited; Not persistent; Memory leaks possible	Active UI state, temporary calculations, session data
SessionStorage	Auto cleanup on close; Simple API; Tab-isolated; Synchronous	Small capacity (5-10MB); Lost on tab close; String-only; Blocks UI	Temporary filters, wizard state, tab-specific data
REST/GraphQL API	Unlimited capacity; Multi-user sync; Centralized control; Backup/recovery	Network dependency; Latency issues; Server costs; Complex infrastructure	Enterprise deployments, collaboration, shared dashboards
Firebase/Supabase	Real-time sync; Built-in auth; Managed infrastructure; Quick setup	Vendor lock-in; Cost at scale; Network dependency; Limited customization	Rapid prototyping, real-time features, small-medium apps
PouchDB + CouchDB	Offline-first; Auto sync; Conflict resolution; Works offline	Learning curve; Specific data model; Sync complexity; Performance overhead	Offline apps, eventual consistency, distributed data
WebRTC	No server needed; Direct P2P; Low latency; Private	Complex setup; Requires signaling; Not persistent; Connection issues	Real-time collaboration, peer sharing, live editing
SQLite WASM	Full SQL support; Relational queries; Transactions; Familiar API	RAM-limited (100s MB); Manual persistence; Larger bundle size; Serialization overhead	Complex queries, relational data, SQL familiarity

Storage Type	Pros	Cons	Best For
DuckDB WASM	Handles GBs of data; OLAP queries; Parquet support; Fast analytics	Large bundle (~10MB); Learning curve; Newer technology; Limited docs	Analytics workloads, large datasets, BI queries
RxDB	Reactive queries; Multi-tab sync; Encryption; Offline-first	Complex setup; Large bundle; Learning curve; Performance overhead	Reactive apps, multi-tab coordination, encrypted data
Gun.js	Decentralized; P2P sync; Offline-first; Graph database	Different paradigm; Learning curve; Limited tooling; Sync complexity	Decentralized apps, graph data, P2P networks
IPFS	Permanent storage; Content-addressed; Decentralized; Immutable	Slower access; Requires gateway; Complex setup; Not for mutable data	Public data sharing, immutable content, archival
Ceramic Network	User-owned data; Cross-app portability; Decentralized identity; Verifiable	Emerging tech; Complex setup; Limited adoption; Infrastructure needs	User-owned configs, cross-app data, Web3 apps

6.8.9 Decision Matrix

Choose IndexedDB when: - You need persistent, structured data storage - Working with dashboards, configs, or extension state - Capacity requirements exceed LocalStorage limits - You need transactions and complex queries

Choose LocalStorage when: - Storing simple key-value preferences - Data size is under 5MB - You need synchronous access - Simplicity is more important than features

Choose In-Memory State when: - Data is temporary and session-specific - Performance is critical - You don't need persistence across refreshes - Working with active UI state

Choose Backend API when: - Multi-user collaboration is required - Data needs to be shared across devices - Centralized control and backup are important - Enterprise features are needed

Choose DuckDB/SQLite WASM when: - Complex analytical queries are required - Working with large datasets (100s MB to GBs) - SQL familiarity is a benefit - In-browser analytics is needed

Choose Decentralized Storage (IPFS/Ceramic/Gun) when: - User data ownership is critical - Decentralization is a core requirement - Building Web3 or P2P applications - Avoiding vendor lock-in is important

6.9 Migration & Versioning

- **Schema Versioning**: Handle data format changes
- **Automatic Migration**: Upgrade old configurations
- **Rollback Support**: Revert to previous versions

- **Cross-Storage Sync**: Coordinate data across storage layers

  Note: Settings management, keybinding, theme, and layout configurations are detailed in Section 2.2.3.

Chapter 7

References & Inspiration

7.1 Open Source Projects

7.1.1 Observable Ecosystem

- Observable Runtime - Reactive notebook runtime with dependency resolution
- Observable Plot - Declarative visualization grammar
- Observable Inputs - Interactive form controls and widgets
- Observable Framework - Static site generator for data apps

7.1.2 tldraw Ecosystem

- tldraw - Infinite canvas SDK with collaborative features
- Signia - Fine-grained reactive state management
- tldraw-yjs - Yjs integration for collaboration

7.1.3 Data & SQL:

- DuckDB WASM - In-browser analytical SQL database
- SQLite WASM - SQLite compiled to WebAssembly
- Arquero - Query processing and transformation library

7.1.4 Collaboration:

- Yjs - CRDT framework for building collaborative applications
- Automerge - JSON-like data structure for collaboration
- Loro - High-performance CRDT library

7.1.5 State Management:

- Zustand - Lightweight state management
- Jotai - Primitive and flexible state management
- Valtio - Proxy-based state management
- Signals - Fine-grained reactivity

7.1.6 Canvas & Rendering

- Konva.js - 2D canvas framework
- Fabric.js - Canvas library with SVG support
- PixiJS - WebGL rendering engine
- rbush - R-tree spatial indexing

7.1.7 UI Component Libraries

- shadcn/ui - Copy-paste components built on Radix UI
- Radix UI - Unstyled, accessible component primitives
- Headless UI - Unstyled, accessible UI components
- Mantine - Full-featured React component library
- Chakra UI - Accessible component library
- Ant Design - Enterprise-grade UI design system
- Material UI - Material Design component library

7.2 Platform Documentation

- Observable Documentation - Notebook concepts and API
- Count.co Documentation - Canvas-based BI platform
- tldraw Developer Docs - Canvas SDK and API reference
- Omni Docs - Documentation platform architecture

7.3 Technical Articles & Resources

7.3.1 Observable

- How Observable Runs - Runtime architecture
- Observable's Not JavaScript - Reactive semantics
- Introduction to Data - Data loading patterns

7.3.2 tldraw

- Building a Collaborative Canvas - Collaboration architecture
- How tldraw Works - Shape system and state management
- Performance Optimization - Rendering optimizations

7.3.3 DuckDB WASM

- DuckDB WASM Performance - Benchmarks and architecture
- In-Browser Analytics - WASM integration guide

7.3.4 Collaboration

- Yjs Documentation - CRDT concepts and API
- CRDT Explained - Conflict-free replicated data types
- Real-time Collaboration Patterns - Figma's approach

7.4 Design Patterns & Architecture

- Reactive Programming - Introduction to reactive thinking
- Flux Architecture - Unidirectional data flow
- Event Sourcing - Event-driven architecture
- CQRS Pattern - Command Query Responsibility Segregation
